**University of Alberta**

# QUANTITATIVE ANALYSIS OF THE SIMD DSP-RAM ARCHITECTURE

by

**Steven James Dillen**

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Fall 2003

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisisitons et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*
*ISBN: 0-612-87749-3*
*Our file Notre référence*
*ISBN: 0-612-87749-3*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

Canada

**University of Alberta**

**Library Release Form**

**Name of Author**: Steven James Dillen

**Title of Thesis**: Quantitative Analysis of the SIMD DSP-RAM Architecture

**Degree**: Master of Science

**Year this Degree Granted**: 2003

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

**Date**: JULY 19, 2003

**University of Alberta**

**Faculty of Graduate Studies and Research**

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Quantitative Analysis of the SIMD DSP-RAM Architecture** submitted by Steven James Dillen in partial fulfillment of the requirements for the degree of **Master of Science**.

Martin J. Jagersand

**Date:** Aug 22, 2003

# Abstract

This dissertation presents a quantitative analysis of the pre-existing DSP-RAM computer architecture. DSP-RAM is a parallel computing architecture that combines an array of simple digital signal processors and a SRAM memory core. This integration exposes, and exploits, the high, internal bandwidth available at the memory's sense amplifiers, allowing DSP-RAM to out perform conventional high-performance microprocessors for moderately-parallel algorithms. Through detailed analysis of the architecture, including area and delay measurements for each sub-block, the design of the processor is refined to achieve a better ratio of performance to area. The modified 50-MHz DSP-RAM out-performed a 1.6-GHz Intel Pentium 4, using SSE2 extensions, for six of the seven algorithms studied, achieving an average performance increase of 2.5 times.

If we knew what it was we were doing,
it would not be called research, would it?

—Albert Einstein

*To Kristy, Kaitlynn and Jessica*

# Acknowledgements

I am grateful for the support provided for this work by the Natural Sciences and Engineering Research Council of Canada, and Micronet R&D.

I would also like to thank Raymond Sung for providing me with many references for all aspects of digital design, and for the many technical discussions we had. Thanks to Amir Alimohammad and Hua Ai, who also worked with the DSP-RAM architecture, and who provided instrumental feedback during the course of this research. Thanks also to John Koob who provided feedback on how to best present the data, and was always available to answer any UNIX, vi, or LaTeX related question. Also, thanks goes out to Tyler Brandon and Dan Leder who were always willing to help solve any problem in one of the CAD tools. I would also like to acknowledge all my fellow graduate students with whom I shared, and enjoyed, many lunch-time and late night conversations.

Special thanks is required for my supervisor, Dr. Bruce Cockburn, who not only provided invaluable suggestions and feedback, but also was an excellent mentor, who helped refine my technical abilities, and taught professionalism through example.

Most of all, I thank my family. Especially my wife, Kristy, and daughters, Kaitlynn and Jessica, for their support, patience and understanding throughout my degree.

# Contents


**1 Introduction** **1**

1.1 Overview . . . . . 1

1.2 Thesis Organization . . . . . 4

**2 Background** **5**

2.1 Parallel Architectures . . . . . 5

2.1.1 Sun Microsystems Visual Instruction Set (VIS™) . . . . . 6

2.1.2 Intel SIMD Extensions . . . . . 8

2.1.3 Motorola AltiVec™ . . . . . 10

2.1.4 SIMD Comparison . . . . . 12

2.1.5 Texas Instruments' (TI) VelociTI™ Architecture . . . . . 13

2.2 Processor-In-Memory Architectures . . . . . 16

2.2.1 Terasys . . . . . 16

2.2.2 Integrated Memory Array Processor (IMAP) . . . . . 16

2.2.3 Vector IRAM . . . . . 17

2.2.4 C•RAM . . . . . 19

2.2.5 Summary of PIM Architectures . . . . . 21

2.3 Image Processing Kernels . . . . . 23

2.3.1 Thresholding . . . . . 24

2.3.2 Sobel Dx and Dy . . . . . 24

2.3.3 Dilate and Erode . . . . . 25

2.3.4 Box and Median Filters . . . . . 26

2.4 Low-Power VLSI Design . . . . . 27

2.4.1 Charging and Discharging Capacitance . . . . . 27

2.4.2 Short Circuit Current . . . . . 28

2.4.3 Leakage Current . . . . . 30

2.4.4 Static Power Dissipation . . . . . 30

2.4.5 Low-power DSP-RAM Architecture . . . . . 31

2.5 Development Platform . . . . . 34

2.5.1 Integrator/AP . . . . . 35

2.5.2 Integrator/CM7TDMI . . . . . 37

2.5.3 Integrator/LM-XCV600E+ . . . . . 37

**3 Evaluation of the Original DSP-RAM Architecture** **40**
3.1 DSP-RAM Architecture . . . . . 42
3.1.1 Networking . . . . . 44
3.1.2 ALU . . . . . 46
3.1.3 Conditional Operation . . . . . 48
3.1.4 Arbiter . . . . . 49
3.2 The C++ Emulator . . . . . 49
3.3 VHDL Implementation . . . . . 51
3.3.1 Shifter . . . . . 52
3.3.2 Stack . . . . . 54
3.3.3 Arbiter . . . . . 56
3.4 Results . . . . . 58
3.4.1 Component Results . . . . . 58
3.4.2 Critical Paths . . . . . 59
3.4.3 PE Results . . . . . 62

**4 Image Processing Experiments on DSP-RAM** **65**
4.1 Thresholding . . . . . 65
4.2 Inter-PE Communication . . . . . 68
4.3 Sobel Dx Operator . . . . . 72
4.4 Dilate . . . . . 74
4.5 Results . . . . . 76

**5 Architecture Modification Experiments** **80**
5.1 Inter-PE Communication . . . . . 80
5.2 Enhanced PE Routing . . . . . 85
5.3 Algorithm Performance . . . . . 87
5.3.1 Threshold . . . . . 87
5.3.2 Inter-PE Communication . . . . . 87
5.3.3 Sobel Dx Operator . . . . . 89
5.3.4 Dilation . . . . . 89
5.3.5 Results . . . . . 90

**6 Future Research Directions** **95**

**7 Discussion and Conclusions** **99**

**Bibliography** **102**

**A DSP-RAM Controller Implementation** **105**
A.1 Register description . . . . . 105
A.2 Example programs . . . . . 107

**B DSP-RAM C++ Emulator Source Code** **112**
B.1 DSP-RAM Class . . . . . 112
B.2 Integer Class . . . . . 135
B.3 Shift Types . . . . . 141

**C DSP-RAM VHDL Source Code** **143**
C.1 AMBA Protocol . . . . . 154
C.2 DSP-RAM PE VHDL . . . . . 184
C.2.1 Logic . . . . . 184
C.2.2 DSP-RAM Processing Element Components . . . . . 192

**D Source Code for the Image Processing Algorithms** **211**
D.1 Pentium 4 Source Code . . . . . 211
D.2 DSP-RAM Source Code . . . . . 218

# List of Tables


2.1 Comparison of Five Commercially-available SIMD Extensions . . . 12
2.2 Comparison of Processor-In-Memory (PIM) Architectures . . . . . 23

3.1 Stack Operation Codes . . . . . 54
3.2 Area and Speed Results for PE Components . . . . . 58
3.3 PE Critical Paths (For Xilinx VirtexE) . . . . . 61
3.4 Area and Speed Results for PE Implementations . . . . . 63

4.1 Inter-PE Communication Pipeline . . . . . 69
4.2 DSP-RAM Image Processing Results (512×512 Image) . . . . . 77

5.1 Area / Speed Results for PE with Enhanced Routing . . . . . 86
5.2 Modified DSP-RAM Image Processing Results (512×512 Image) . 92

A.1 DSP-RAM Controller Registers . . . . . 106
A.2 Control Word Register Definition . . . . . 108
A.3 Shift Modes . . . . . 109
A.4 Shift Modes . . . . . 109

# List of Figures


1.1 A SIMD architecture . . . . . 2

2.1 Different data representations for a single register . . . . . 6
2.2 SIMD addition of packed registers . . . . . 7
2.3 TMS320C6000 series block diagram [35] . . . . . 13
2.4 Instruction packing on the VelociTI architecture . . . . . 15
2.5 VIRAM-1 **vhalf** $vr1, $vr0 instruction . . . . . 18
2.6 C•RAM processing element (PE) with interconnect . . . . . 20
2.7 Interior pixel and neighbourhood . . . . . 24
2.8 Effects of applying Sobel gradients on an image . . . . . 25
2.9 Effects of dilate and erode on an image . . . . . 26
2.10 Effects of noise filtering on an image . . . . . 27
2.11 Static CMOS inverter . . . . . 28
2.12 Short circuit current in a static CMOS inverter . . . . . 29
2.13 Pseudo NMOS 4-input NOR gate . . . . . 31
2.14 Minimum required operating voltage versus frequency . . . . . 33
2.15 Power savings in parallel processor systems . . . . . 33
2.16 ARM Integrator series rapid prototyping platform [10] . . . . . 35
2.17 Integrator/AP motherboard [7] . . . . . 36
2.18 Integrator/CM7TDMI microprocessor core module [5] . . . . . 38
2.19 Integrator/LM-XCV600E+ logic module [6] . . . . . 38

3.1 SIMD 16-bit addition performance . . . . . 41
3.2 SIMD 16-bit multiplication performance . . . . . 42
3.3 DSP-RAM processing element . . . . . 43
3.4 Broadcast bus to memory pipeline . . . . . 44
3.5 Data path used to shift data through the PE array . . . . . 45
3.6 Data stored incorrectly through the shift bus . . . . . 47
3.7 ALU pipeline . . . . . 48
3.8 DSP-RAM design flow . . . . . 50
3.9 Shifter circuit . . . . . 53
3.10 Stack circuit . . . . . 54
3.11 Stack element circuit . . . . . 55
3.12 Arbiter circuit . . . . . 56
3.13 Arbiter controller state diagram . . . . . 57
3.14 Critical paths in the processing element . . . . . 60

3.15 Digital circuit delays . . . . . 61

4.1 Threshold operation on DSP-RAM . . . . . 67
4.2 Image width spans across four memory rows . . . . . 69
4.3 DSP-RAM controller stores and injects pixels from edge PEs . . . . 71
4.4 The Sobel operation on DSP-RAM . . . . . 73
4.5 Computation time versus image size . . . . . 79

5.1 Memory transfer between PEs . . . . . 81
5.2 Memory transfer between PEs modified to use a 16-bit shift bus . . 82
5.3 Pre-computed left or right shift bus selection . . . . . 83
5.4 Accumulator registers selection . . . . . 83
5.5 FIR filter . . . . . 84
5.6 ALU by-pass routing . . . . . 86
5.7 Performance improvements over the original DSP-RAM architecture 91
5.8 Computation time versus image size . . . . . 93
5.9 Median filter performance for varying image sizes . . . . . 94
5.10 DSP-RAM generated images . . . . . 94

6.1 Register-register architecture . . . . . 97

A.1 Control register . . . . . 105

# Listings


2.1 if-then-else control structure . . . . . 14
4.1 DSP-RAM data hazard . . . . . 76
A.2 DSP-RAM FIR Filter . . . . . 109
A.1 Load DSP-RAM memory . . . . . 109
B.1 DSP-RAM Class Definition . . . . . 112
B.2 DSP-RAM Class Implementation . . . . . 115
B.3 Integer Classes Definition . . . . . 135
B.4 Integer Classes Implementation . . . . . 136
B.5 Shift Types Definition . . . . . 141
C.1 DSP-RAM System . . . . . 143
C.2 PE Array . . . . . 146
C.3 RAM Column . . . . . 147
C.4 RAM . . . . . 149
C.5 Controller . . . . . 150
C.6 AHB to APB Interface . . . . . 154
C.7 AHB to AHB Top Level . . . . . 161
C.8 AHB to APB System . . . . . 166
C.9 AHB Address Decoder . . . . . 170
C.10 AHB Mux . . . . . 172
C.11 APB Registers . . . . . 174
C.12 APB Interrupt Controller . . . . . 178
C.13 AHB SRAM Interface . . . . . 181
C.14 Logic Package . . . . . 184
C.15 Bidirectional Driver . . . . . 186
C.16 16-Bit Decoder . . . . . 187
C.17 D Flip-Flop . . . . . 187
C.18 16-1 MUX . . . . . 188
C.19 8-1 MUX . . . . . 189
C.20 4-1 MUX . . . . . 190
C.21 2-1 MUX . . . . . 191
C.22 Register . . . . . 191
C.23 Tri-state Buffer . . . . . 192
C.24 ALU . . . . . 192
C.25 Arbiter . . . . . 194
C.26 Byte Select . . . . . 196

C.27 Shifter . . . . . 197
C.28 Stack Element . . . . . 200
C.29 Stack . . . . . 201
C.30 PE . . . . . 203
D.1 Pentium 4 Source Code . . . . . 211
D.2 Image Class Definition . . . . . 218
D.3 Image Class Implementation . . . . . 219
D.4 Threshold . . . . . 243
D.5 Sobel Dx and Dy . . . . . 244
D.6 Box Filter . . . . . 248
D.7 Dilate . . . . . 251
D.8 Erode . . . . . 254
D.9 Median Filter . . . . . 257

# List of Symbols

| | |
|---|---|
| **ALU** | Arithmetic Logic Unit |
| **AMBA** | ARM Microcontroller Bus Architecture |
| **ASIC** | Application-Specific Integrated Circuit |
| **CLB** | Configurable Logic Block |
| **CPU** | Central Processing Unit |
| **C•RAM** | Computational Random-Access Memory |
| **DMA** | Direct Memory Access |
| **DRAM** | Dynamic Random-Access Memory |
| **DSP** | Digital Signal Processing |
| **FPGA** | Field-Programmable Gate Array |
| **FPU** | Floating Point Unit |
| **IC** | Integrated Circuit |
| **ISA** | Instruction Set Architecture |
| **LED** | Light Emitting Diode |
| **PCB** | Printed Circuit Board |
| **PCI** | Peripheral Component Interconnect |
| **PDA** | Personal Data Assistant |
| **PE** | Processing Element |
| **PIM** | Processors-in-Memory |
| **RAM** | Random-Access Memory |
| **RAW** | Read After Write |
| **SAD** | Sum of Absolute Differences |
| **SDRAM** | Synchronous Dynamic Random-Access Memory |
| **SIMD** | Single Instruction Stream, Multiple Data Streams |
| **SOC** | System-on-Chip |
| **SORC** | System-on-Reconfigurable-Chip |
| **SRAM** | Static Random-Access Memory |
| **SSRAM** | Synchronous Static Random-Access Memory |
| **SSD** | Sum of Squared Differences |
| **SSE** | Streaming SIMD Extensions |
| **UART** | Universal Asynchronous Receiver/Transmitter |
| **VHDL** | VHSIC Hardware Description Language |
| **VHSIC** | Very High Speed Integrated Circuits |
| **VIS** | Visual Instruction Set |
| **VLIW** | Very Long Instruction Word |

# Chapter 1

# Introduction

## 1.1 Overview

Computer vision applications require a system that is capable of handling the massive computational requirements associated with image processing. Typically, computer vision applications require a high-performance microprocessor system. For applications like mobile robots, however, these power-hungry processors are not acceptable. For example, a mobile robot must carry its own power source and can only operate as long as there is enough power remaining. Such robots cannot afford the space, weight, or power requirements of a high-end microprocessor in the system, but still require sufficient performance to host the necessary computer vision applications. Similar requirements exist for portable hand-held devices such as cellular telephones and personal data assistants (PDAs). Thus portable computer vision systems require a different approach. DSP-RAM [42] [22] [12] is a parallel single instruction stream, multiple data streams (SIMD) architecture (Figure 1.1) that provides an alternative platform to a high-performance microprocessor.

DSP-RAM achieves high performance from its moderately-parallel, processors-in-memory (PIM) style of architecture [13] [15] [21]. By integrating many simple Digital Signal Processors (DSPs) with a memory core, DSP-RAM is able to utilize the high data bandwidth available at the memory's sense amplifiers. For computer vision applications, this means that each DSP can simultaneously execute the same image processing algorithm on separate streams of pixel data. This results in an instruction count, and therefore a CPU execution time [17], for the entire application

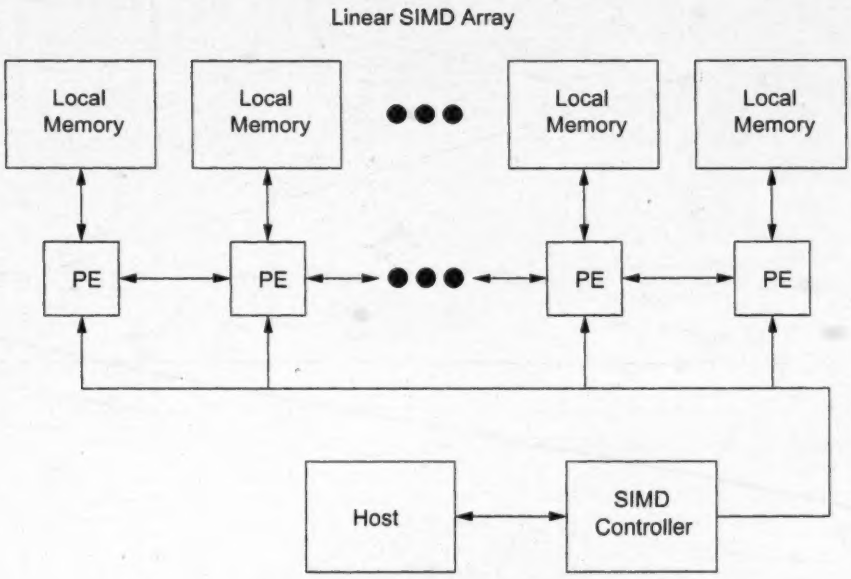



Figure 1.1: A SIMD architecture

that is inversely proportional to the number of processors. This research project has demonstrated that DSP-RAM can match the performance of a high-end microprocessor, while operating at a lower clock frequency, because of the high-bandwidth bus available to the memory core as well as the larger number of processing units.

PIM-style architectures also have the benefit of reduced power consumption. This reduction originates from two sources: a lower clock frequency and minimizing the driving of data off-chip. The lower clock frequency alone does not lower the power dissipation since greater parallelism usually implies higher capacitance. However, relaxed timing requirements (i.e., a longer clock period) allows the system to use a lower operating voltage and still meet all the timing requirements for the system [9] [44]. Since power consumption in digital CMOS is proportional to voltage squared, substantial power savings may be achieved. The second source of power savings is due to the processor being integrated into the same chip as the memory core. This means that data transfers between the memory and the processors remain inside the same integrated circuit (IC). Note that off-chip data bus wires have capacitances that are typically in the picoFarad (pF) range whereas the bus wires inside the IC fall in the femtoFarad (fF) range [38]. These two power

reduction techniques allow DSP-RAM to be an attractive solution for low-power systems. The power issues are considered in more detail in Section 2.4.

This thesis evaluates the ability of the DSP-RAM architecture to meet the growing demand of high-performance image processing in a low-power embedded system, such as a mobile robot. To accomplish this analysis, the DSP-RAM processing element (PE), the network structure between PEs and the data organization within a PE are evaluated with several image processing kernels. By isolating the bottlenecks in the system for each algorithm, modifications to the system were made to improve the original DSP-RAM architecture. For each of these modifications, the algorithms had to be re-coded in order to utilize the new hardware. In addition to architecture modifications that increase algorithm performance, modifications that decrease the area of the PE were also explored. As stated earlier, the number of processors has a direct impact on the CPU execution time. By increasing the number of processors that can be implemented on a single IC, the execution time of the algorithm will be decreased. The modifications that were done to decrease the area of a PE were independent of the algorithms and are set through generic parameters in a VHDL model of DSP-RAM.

To gain further insight into architecture and area trade-offs, a synthesizable VHDL model was designed and implemented for the Xilinx Virtex 2000E FPGA. FPGA technology is an attractive platform for prototyping SIMD architectures that follow the PIM methodology. Today's FPGAs integrate both logic cells and memory blocks and now provide multiple millions of equivalent logic gates and megabytes of memory within one device [41] [40] [39]. For example, in one synthesized configuration, a DSP-RAM system with 32 processors was implemented with each processor containing one kilobyte (kB) of local memory.

The VHDL model is also used to compare DSP-RAM implementations in different FPGA technologies. By synthesizing the same VHDL model for different devices, area and operating frequencies could be compared. Recently available FPGAs, such as the Xilinx Virtex II Pro, offer configurable (up to 18x18) optimized integer multipliers that allow more PEs to be implemented inside a single device. By basing a SIMD system on FPGA technology, more processors and more local

memory will become implementable on one IC as FPGA technology advances following Moore's law and produces devices with larger gate counts. Using a parameterized and synthesizable architecture means that the engineering costs in moving to new technologies will be minimized. This contrasts with the rapidly rising costs of custom integrated circuit designs in the latest available technology.

From our findings, we propose several modifications to the DSP-RAM architecture that improve the performance of the selected image processing kernels. The performance improvements arose from simplifications that decreased the area of the PE, thus allowing more PEs to be synthesized on a single device. Also, performance could be improved through modifications that removed bottlenecks in the kernels themselves. Our research shows that a DSP-RAM co-processor can achieve performance levels similar to high-performance desktop workstations and still offer an attractive low-power solution to an embedded system.

## 1.2 Thesis Organization

This thesis is organized into seven chapters. Chapter 2 discusses background material related to: parallel processing in both commercial and research systems; the image processing kernels used in the research; low-power VLSI design; and the implementation platform that we used. Chapter 3 outlines the DSP-RAM architecture and presents speed and area results for different synthesized versions. The designs of several image processing kernels, along with the results obtained on the original architecture are then discussed in Chapter 4. Chapter 5 presents the modifications made to the architecture and discusses the benefits and drawbacks for each of them. Chapter 6 proposes further research, based on DSP-RAM, to further increase performance, and alleviate some of the problem areas. Chapter 7 ends the thesis with a discussion of the major conclusions and contributions made by this research.

# Chapter 2

# Background

This chapter briefly surveys parallel architectures used in today's commercial microprocessors, as well as research SIMD architectures. The image processing kernels used in our research are also described along with an explanation of how to map the algorithms onto a SIMD architecture. Next, low-power VLSI design techniques are outlined with an emphasis on how power dissipation is reduced in the DSP-RAM architecture. The chapter concludes with a description of the FPGA and microprocessor architecture that was used in this project: the System-on-a-Chip (SOC) Rapid Prototyping Platform (RPP).

## 2.1 Parallel Architectures

The majority of today's high-performance microprocessors include SIMD instructions in order to maximize performance for data parallel applications [11]. These architectural features exploit the fact that for many applications, the majority of the variables are smaller than the width of the registers. By utilizing the same wide data path, multiple values can be passed in parallel. These data values are stored in a *packed register*, as illustrated in Figure 2.1. Using the same 64-bit register, the data can be either a single 64-bit value in the native full precision of the processor, two 32-bit values, four 16-bit values, or eight 8-bit values. In this example, the latter three formats are SIMD representations. During instruction execution, each separate data value undergoes the same operation in parallel and the results are stored in a packed destination register. Figure 2.2 illustrates how four 16-bit numbers are

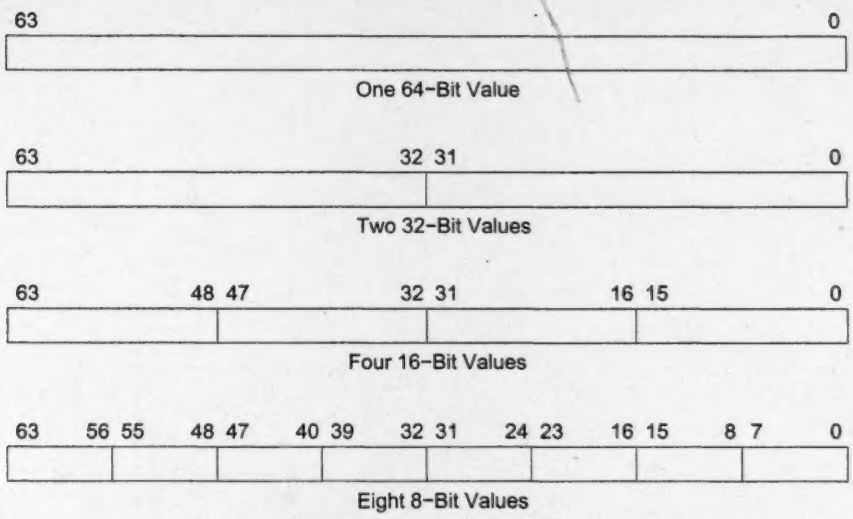



Figure 2.1: Different data representations for a single register

added together in parallel. Registers A and B contain four signed 16-bit numbers packed into 64-bit registers. When performing a SIMD addition, the ALU hardware is reconfigured to treat the 64-bit registers as packed registers and will add the four pairs of data in parallel.

The remainder of this section examines the SIMD architectures found in Sun Microsystem's UltraSPARC, Intel's Pentium 4, and Motorola's PowerPC. AMD's Direct3D technology and HP MAX also offer SIMD extensions, but are not covered in this section. In addition to SIMD extensions to existing processor architectures, the Texas Instruments (TI) High VelociTI processing architecture is described. The High VelociTI architecture is a Very Long Instruction Word (VLIW) processor with the entire microprocessor design focused on exploiting parallelism.

### 2.1.1 Sun Microsystems Visual Instruction Set (VIS™)

Sun Microsystems' Visual Instruction Set (VIS) was one of the first SIMD extensions to a general-purpose processor's instruction set. It was introduced in 1995 in the UltraSPARC™ processor. The VIS extensions were designed to enhance graphics performance [37], but have also been used to improve performance for applications in other areas such as networking and telecommunications [33]. Although

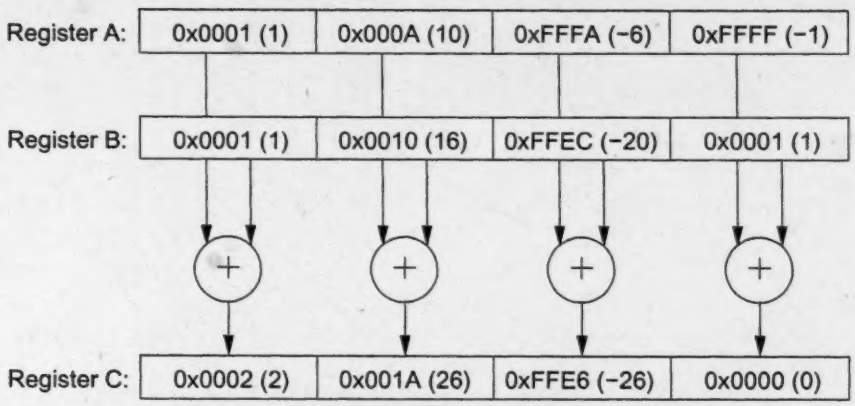



Figure 2.2: SIMD addition of packed registers

VIS supports only integer processing, it is incorporated into the two floating-point pipelines of the UltraSPARC processor. This integration keeps the regular integer unit and the VIS unit in separate pipelines, thereby allowing both an integer instruction and two VIS instructions to be executed in parallel. VIS shares the 32 64-bit registers in the floating-point register file. While the register file is shared, the VIS datapath is independent from the FPU datapath. Therefore the UltraSPARC is able to concurrently issue one floating-point instruction and one VIS instruction, subject to certain grouping criteria [37].

The SIMD unit operates on multiple data values packed into a single 64-bit register. VIS supports five different methods of organizing the packed data:

1. Four unsigned 8-bit numbers
2. Two signed 16-bit numbers
3. Two signed 32-bit numbers
4. Four signed 16-bit numbers
5. Eight unsigned 8-bit numbers

The first two data representations only use 32 bits of the register. Combining eight unsigned 8-bit numbers is not supported for the arithmetic and logical operations. This data organization is used with data access and with the **pdist** instruction. The

**pdist** instruction computes the sum of absolute differences (SAD) and is commonly required for many video encoding techniques [31]. Arithmetic and logical operations only support four 8-bit numbers because VIS was originally designed to enhance the performance of graphics operations and the required data grouping primarily consisted of the four 8-bit pixel intensities $\alpha$, R, G, and B. This design decision adversely affects the performance of other algorithms in two significant ways. First, since eight-way data parallelism is not supported for the arithmetic and logical operations, the performance improvement of algorithms that contain many parallel 8-bit data operations is limited. Second, since one of the operands was assumed to be primarily 8-bit data, the multiply instructions only support 8-bit by 16-bit multiplications. Therefore to compute a 16-bit by 16-bit multiplication requires three separate instructions, which increases instruction count, register pressure and data dependancies for 16-bit processing [31].

VIS, like the original SPARC instruction set architecture (ISA), uses a three-address instruction set. Two of the addresses represent operand registers and the third address is the result register. A three-address instruction set generally requires less instructions to implement a given code sequence than a two-address instruction set [32].

### 2.1.2 Intel SIMD Extensions

Intel first introduced SIMD capabilities into the 32-bit Pentium® architecture with the MMX instruction set extention in 1996 [23]. The Pentium III introduced the Streaming SIMD Extensions (SSE) while still retaining the existing MMX technology [27]. Intel augmented the existing MMX and SSE instructions by introducing SSE2 instructions with the Pentium 4 and Xeon® processors [19]. We will now discuss each of these technologies and the successive enhancements that were made to the SIMD instructions in each generation of processor.

#### MMX™

MMX technology is very similar to Sun's VIS extensions to the UltraSPARC. The MMX registers alias the register file for the FPU. Unlike VIS, Intel does not allow

both the SIMD unit and the FPU to be used simultaneously. Instead, a high latency state change is required to switch between operating modes [19] [33].

MMX uses multiple data values packed into eight 64-bit registers. This allows for the following data arrangements:

1. Eight packed 8-bit integers (Bytes)
2. Four packed 16-bit integers (Words)
3. Two packed 32-bit integers (Doublewords)
4. One 64-bit integer (Quadword)

Arithmetic and logical operations are available for all data organizations.

MMX does not contain any operations supporting the sum of absolute differences or any other operation on data elements packed into a single register, such as minimum, maximum, or average value. If these operations are required, the data must be unpacked and processed by the integer unit in a sequential format.

MMX uses a two-address instruction format where one of the source registers also acts as the destination register. A two-address instruction set results in simpler instruction decode circuitry and fewer bits in the instruction word at the expense of more complex code. Code complexity results from additional move instructions that are required to avoid altering the original data [32].

**SSE**

SSE technology extended MMX in two ways: it added support for SIMD operations on packed single-precision floating-point values; it also extended integer operations to include operations involving elements packed into the same register. To support SIMD floating-point operations, eight 128-bit registers (XMM0-XMM7) were added to the architecture. The XMM registers are able to pack four single-precision floating-point values together. Unlike the MMX registers, the XMM registers are independent from the other register files in the CPU so the FPU can be used independently from the floating-point SSE instructions. The MMX registers remain

aliased to the floating-point register file so the SIMD integer operations are not independent from the FPU.

SSE also extended the SIMD integer functionality beyond MMX to include computing the sum of absolute differences between two packed registers, averaging the values in a single packed register, and finding the maximum and minimum of the values in a packed register.

**SSE2**

SSE2 instructions added the ability to perform integer SIMD operations on the XMM register set. With the new extensions, the following data arrangements are possible with the XMM registers:

1. Four single-precision floating-point values
2. Two double-precision floating-point values
3. Sixteen 8-bit integers (Bytes)
4. Eight 16-bit integers (Words)
5. Four 32-bit integers (Doublewords)
6. Two 64-bit integers (Quadwords)
7. One 128-bit integer (Double Quadword)

These new data organizations provide twice the parallelism for integer operations without any additional registers. Another benefit from this is that the FPU can now be utilized in parallel with SSE2 integer SIMD instructions whereas the MMX and SSE extensions required a separate state change. SSE2 also added the ability to operate on two double-precision floating-point values in parallel.

### 2.1.3 Motorola AltiVec™

Motorola introduced the AltiVec technology as SIMD extensions to the PowerPC architecture [3]. AltiVec technology is implemented as a separate functional unit

called the Vector Unit (VU). The VU is thus orthogonal to the FPU and the Integer Unit. It consists of a 128-bit register file that contains 32 registers. Each register can pack data in the following formats:

1. Four single-precision floating-point values
2. Sixteen 8-bit integers (Bytes)
3. Eight 16-bit integers (Half Words)
4. Four 32-bit integers (Words)
5. One 128-bit integer (Quad Word)

AltiVec divides the instructions into two classes: Intra-Element and Inter-Element [14]. Intra-Element instructions perform independent computations on the packed elements, as in the example shown in Figure 2.2. Inter-Element operations are operations that use the elements of a single packed register as operands. Examples of Inter-Element instructions in AltiVec are the "sum across" and "sum of products" operations. The "sum across" instruction adds all data elements in a packed register and stores the single result to a destination register. While AltiVec does not offer a sum of absolute differences instruction, in the set Inter-Element operations, the instruction can be synthesized from other existing instructions in the vector unit [31]. It is possible that by utilizing the sum of products instructions, a sum of squared differences (SSD) alternative to the sum of absolute differences could be used.

AltiVec technology provides both three-address and four-address instructions. Four-address instructions allow multiple arithmetic operations to be integrated into one instruction, while at the same time maintaining a high degree of programming flexibility. An example of a four-address instruction would be the following multiply-add instruction:

**vmaddfp** vD, vA, vC, vB

Registers *vA* and *vC* contain the source operands for the multiply operation. Each product is then added to the corresponding element packed into register *vB*. The

Table 2.1: Comparison of Five Commercially-available SIMD Extensions

| | VIS | MMX | SSE | SSE2 | AltiVec |
|---|---|---|---|---|---|
| Register Width | 64 | 64 | 128 | 128 | 128 |
| Single-Precision FP | No | No | Yes | Yes | Yes |
| Double-Precision FP | No | No | No | Yes | No |
| # of Registers | 32 | 8 | 8 | 8 | 32 |
| SAD Instruction | Yes | No | Yes | Yes | No[a] |
| # of Instructions | 121 | 47 | 62 | 69 | 162 |
| Instruction Type | 3-Addr | 2-Addr | 2-Addr | 2-Addr | 4-Addr |

[a]The SAD instruction can be constructed from other instructions

result for each vector element of the multiply-add instruction is then stored into register *vD*.

### 2.1.4 SIMD Comparison

Sun Microsystem's VIS and Intel MMX are the oldest of the SIMD extensions that we will survey. Both architectures are strictly integer-based and utilize the 64-bit register file of the FPU. VIS could utilize one of the FPU pipelines in parallel with a SIMD instruction, whereas MMX required a state change first. Intel introduced the SSE instructions to extend the SIMD operations to single-precision floating-point. SSE instructions make use of a new 128-bit wide register file named XMM, which can pack together four floating-point values. The main improvement to the integer SIMD operations was the introduction of a sum of absolute differences instruction. SSE2 provided the most beneficial SIMD extensions for Intel. SSE2 extended the use of the 128-bit XMM registers for integer SIMD operations. This doubled the parallelism as well as removed contention on the FPU register file. SSE2 is also the only architecture that offers some support for SIMD operations on double-precision floating-point numbers. Motorola's AltiVec technology was originally architected as a separate vector unit. AltiVec started with a 128-bit register file which supported both integer and single-precision floating-point numbers. AltiVec also offers several inter-element operations for computing the sum across and minimum/maximum operations. Some of the more important features for each architecture are summarized in Table 2.1.

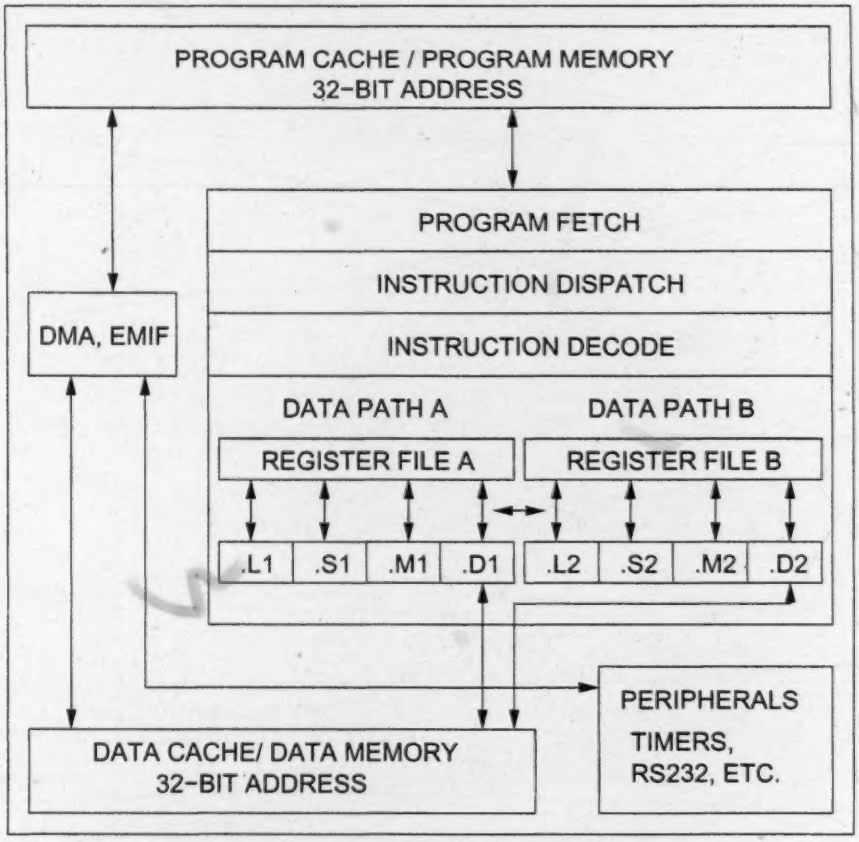



Figure 2.3: TMS320C6000 series block diagram [35]

### 2.1.5 Texas Instruments' (TI) VelociTI™ Architecture

TI took a different approach to improving performance on its TMS320C6xxx series DSPs. Rather than adding a SIMD vector processing unit to the integer CPU, TI changed the entire CPU into a very long instruction word (VLIW) architecture called VelociTI [29] [35]. A block diagram showing the VelociTI architecture of the TMS320C6000 series DSPs appears in Figure 2.3. The VelociTI architecture consists of eight independent functional units and supports the simultaneous fetch, decode and dispatch of up to eight instructions. The architecture is separated into two identical data paths. Each data path maintains sixteen, 32-bit wide registers. The register files each have 10 read ports and 6 write ports with a crossing route in each data path to read operands from the other file. Each data path contains four

Listing 2.1: if-then-else control structure

```
[A1]  ADD .L1 A2,A3,A4   ; if (A1) { A4 = A2 + A3 }
[!A1] SUB .L1 A2,A3,A4   ; else    { A4 = A2 - A3 }
```

functional units (L, S, M, D) so that every instruction can be executed on at least two function units (one from each data path). More common instructions, such as ADD, can be issued to six of the eight functional units. In the TMS320C64x architecture, the functional units were designed to include SIMD-style features. This increases the parallelism by allowing data in the register file to be represented as four 8-bit values, two 16-bit values, or one 32-bit value.

Another feature in the VelociTI architecture is that every instruction supports conditional execution. Conditional execution of instructions is a method of reducing or eliminating branches in simple code sequences [17]. They convert the control dependency of a branch instruction into a data dependency by changing the instruction behaviour depending on a condition stored in a register. Consider the code sequence in Listing 2.1, which illustrates how a simple if-then-else control structure can be coded using conditional instructions. Notice how the control structure has been converted to a data dependency on register A1. All instructions in the VelociTI architecture begin execution. Instructions whose enabling condition is not met by the end of the first phase will not commit results back to the register file. Conditional load and store instructions are canceled before access to data memory is required. This prevents undesired accesses to memory, including memory-mapped peripherals where simply accessing the memory may have undesired side-effects. A downside to annulling an instruction early in the pipeline is that the value of the controlling register must be known at that stage, which may require stalling the pipeline.

The VelociTI architecture also uses an instruction packing technique in order to reduce the code size. In typical VLIW machines, instructions are ordered according to the functional unit they are assigned to [29]. If a functional unit cannot be utilized, it is given a no-operation (NOP) instruction. These NOP instructions are

| Instruction | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| p-bit | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(a) Fully Serial Instructions

| Instruction | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| p-bit | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |

(b) Mixed Parallel Instructions

| Instruction | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| p-bit | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

(c) Fully Parallel Instructions

Figure 2.4: Instruction packing on the VelociTI architecture

stored in the instruction memory, and must be fetched and decoded along with the other instructions. TI does not restrict the instructions stored in program memory to correspond to functional units. Instead, a parallel bit (p-bit) is assigned to each instruction. If set, the bit indicates that the instruction starts execution in parallel with the next instruction. Figure 2.4 shows how different groupings of instructions may be packed on the VelociTI architecture. Figure 2.4(a) shows how the p-bit would be set for a group of 8 instructions labeled A-H. In this case, since no instruction can be operated on in parallel with the next instruction, all the p-bits are zero. Figure 2.4(b) illustrates how parallel instructions can be grouped with some serial instructions. Instructions A, B, C, and D can all be issued in parallel, as can instructions E, F, and G. However, three separate groups of instructions must be issued since instruction D cannot operate in parallel with E, and instruction G cannot operate in parallel with H. Finally, Figure 2.4(c) shows p-bit assignments for the case of eight fully parallel instructions.

## 2.2 Processor-In-Memory Architectures

PIM-style architectures have traditionally involved integrating simple processors within a memory core. In this section we review four representative architectures: Terasys, IMAP, Vector I-RAM and C•RAM.

### 2.2.1 Terasys

Researchers at the Supercomputing Research Center (SRC) developed Terasys, a massively-parallel SIMD processing array in the early 1990s [15]. The SIMD array comprises 32K bit-serial processing elements (PE). Each PE has access to a 2-Kbit column of SRAM memory. The PE operates on three 1-bit registers (A, B, and C) and is limited to the following logical operations:

- A xor B xor C
- AB or AC or BC
- (A xor B) or C

The results of these operations can be stored back to memory, or saved back to the registers.

The communication network for the processing array consists of a global OR network, a partitioned OR, and a parallel prefix network. The global OR simply OR's the signals from all the processors and returns the single bit value to the host. The partitioned OR network divides the processing array into separate groups and was primarily targeted at matrix pivoting operations. The parallel prefix network consists of hardware support for 10 programmable levels which can shift data to the left or to the right a distance of $2^l$, where $l$ represents the programmed level. For example at level 0, data is shifted to a processor's immediate left or right neighbour.

### 2.2.2 Integrated Memory Array Processor (IMAP)

The Integrated Memory Array Processor (IMAP) consisting of 64 PEs was developed at NEC for image processing applications in the early 1990s [43]. The ALU

used in IMAP consists of an 8-bit carry look-ahead adder. Multiplication is implemented using a table look-up method and a shift unit. Each PE maintains twelve general-purpose registers; two shift registers for image data input and output; an inter-PE register for transferring data between neighbour PEs; and a memory register for loading and storing data. IMAP also supports if-then-else control structures through a 1-bit mask register to set individual PEs as active or inactive.

Two PEs share a single memory block in the IMAP system. To avoid reducing performance, the memory system is clocked at twice the frequency of the PEs. The odd-numbered PEs access memory on the first cycle, and the even-numbered PEs access memory on the second cycle. The memory interface is also only four bits wide so two load instructions are required to load a register with an 8-bit value.

In the mid-1990s, NEC described an enhanced successor to IMAP called PIP-RAM which integrates 128 PEs with a 16-Mbit DRAM core [1].

### 2.2.3 Vector IRAM

A vector processor merged with embedded DRAM, called VIRAM-1, was developed around the turn of the century at the University of California at Berkeley (UCB) [21] [20]. VIRAM uses the concept of vector lanes to increase design modularity. Each vector lane consists of two 64-bit ALUs, access to 64-bits of the vector register file, and a memory interface. Each ALU can operate on a single 64-bit data value, two 32-bit data values, or four 16-bit values. The number of elements contained in a vector is under compiler control through the Virtual Processor Width (VPW) register. The first ALU supports integer, fixed-point, and single-precision floating-point operations. Due to area constraints, the second ALU does not include floating-point hardware nor an integer multiplier. Instead, it implements permutation primitives to support inter-element operations. An example permutation primitive is *vhalf*, illustrated in Figure 2.5, which sums the elements in a single vector register. This figure shows that the number of elements stored in $vr0, four in this example, is divided in half. The last half of the elements are stored in $vr1. The vhalf instruction then adds the first half of the elements in $vr0 to the last half, now stored in $vr1. The resulting two elements are stored back into $vr0. By per-

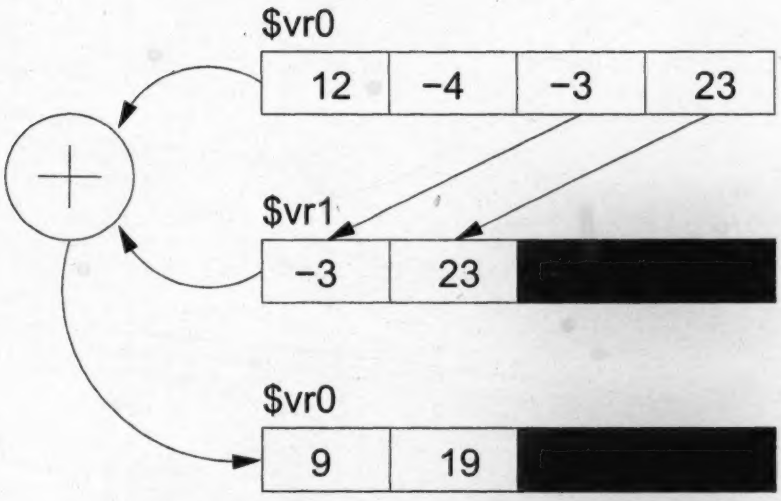



Figure 2.5: VIRAM-1 **vhalf** $vr1, $vr0 instruction

forming the vhalf instruction on $vr0 again, the sum of all four elements would be computed. This time, the number of elements is two, so the second element, 19 in Figure 2.5 would be stored in $vr1. The first element would then be added to the second element and stored back into $vr0, producing the final result of 28—the sum of the original four numbers.

For conditional execution, VIRAM uses a vector flag register and each processor controls a single bit. The results for an operation are only written back to the register file if the corresponding bit in the flag register is set.

Since the memory in the VIRAM system is also intended to be used as main memory for the embedded application, the single-chip implementation is based on an embedded DRAM process. Embedded DRAM has much higher capacity than SRAM, but at a cost of longer latency and the cost of any process modifications required to produce DRAM capacitors and low-leakage access transistors necessary for the DRAM cell. VIRAM maintains eight independently addressed banks of DRAM to help increase the performance of the memory system. The performance increase is primarily due to three features of a multi-bank system: parallel memory access, low access latency, and data caching. Treating each memory bank indepen-

dently allows VIRAM to access different memory locations simultaneously. This structure also allows the host controller to access memory at the same time as VIRAM. Since each individual memory access uses only the circuitry from a single memory bank, the access latency is reduced to below that of a single-bank organization. Finally, a multi-bank DRAM introduces a caching effect. As each bank accesses a memory location, the row of bits is transferred to the sense-amplifiers of the bank. This data row is called "open" because any subsequent accesses to the same data can be served directly from the sense amplifiers with a shorter access time [26]. Multiple memory banks increase the number of open rows available to the system.

VIRAM goes a step further and divides each of the eight independent DRAM banks into sub-banks to increase the number of open rows available. Each sub-bank shares a common data bus, but the control circuitry is sent through pipeline registers inside each sub-bank. Therefore, while only a single data access can be issued at each clock cycle to a DRAM bank, each sub-bank can be accessed independently in subsequent clock cycles in a pipelined fashion.

The VIRAM-1 chip also includes an embedded MIPS scalar processor that issues vector instructions to the VIRAM as well as executes scalar portions of algorithms. The memory system is shared between the scalar and vector processors.

### 2.2.4 C•RAM

C•RAM integrates many bit-serial processors at the memory sense amplifiers of a DRAM [13]. C•RAM was designed to be able to replace commodity DRAM. This added requirements such as redundancy, cost, and area to the design of the PE. Figure 2.6 shows a C•RAM PE including the available interconnect. The ALU is an 8-1 multiplexor (MUX). The three inputs to the ALU are the X register, the Y register, and the memory data. These inputs are connected to the multiplexor's select lines. The truth table of the instruction is sent to the eight inputs to the MUX giving a total of 256 unique instructions. The result of the ALU operation can be stored in the X register, the Y register, the write-enable register, or back into memory.

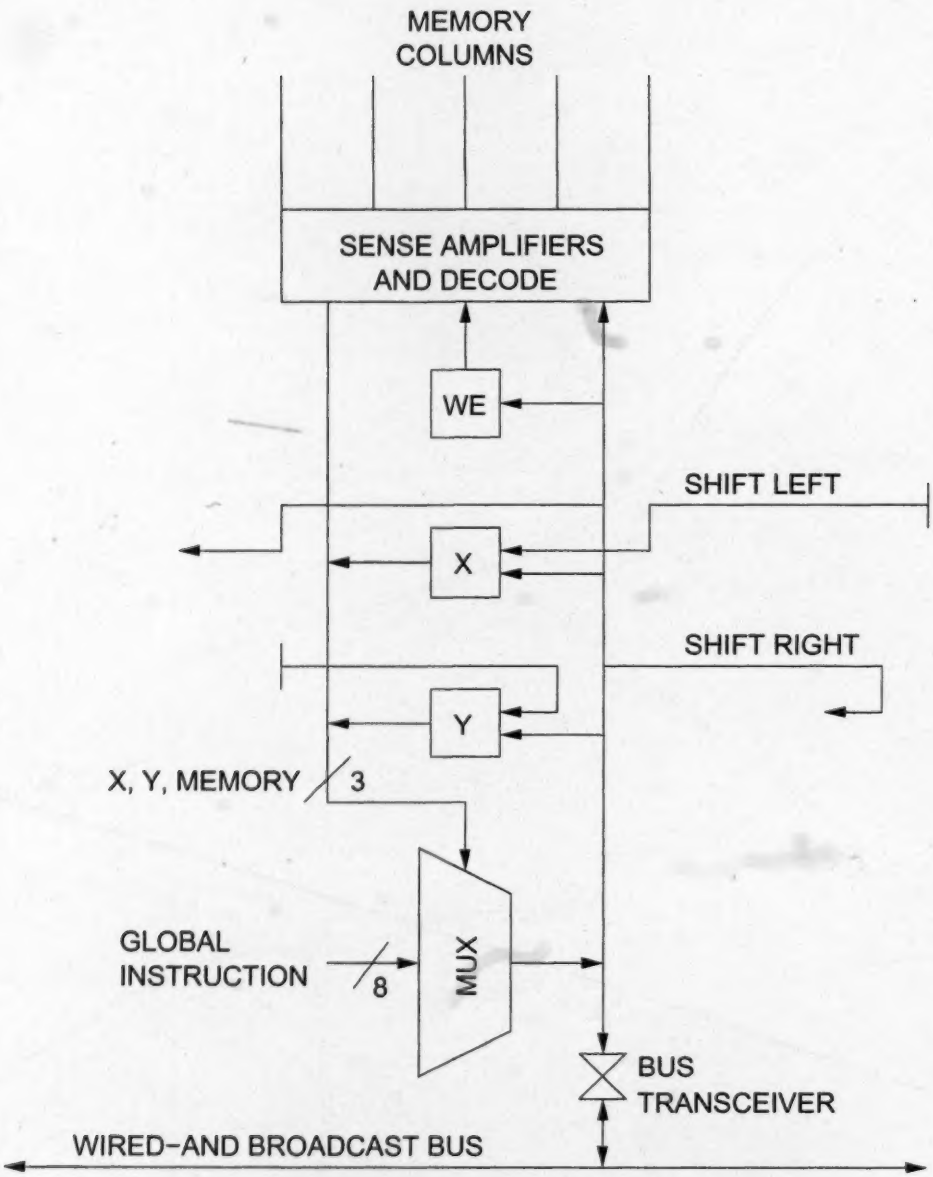



Figure 2.6: C•RAM processing element (PE) with interconnect

C•RAM has a linear network for inter-PE communication. Each processor can shift the result of an instruction to the right and store it in that neighbour's Y register, or shift it left and store it in that neighbour's X register. Two-dimensional communication is supported at the array or chip levels. Thus 2-D communication is simply performed by a sequence of left or right shifts. Once the data reaches the end of the array or the chip, rather than continuing in the left or right direction, the data is shifted to the array above or below it. The final communication structure available is the broadcast bus. The broadcast bus can be used to broadcast a single value to all PEs or can be used to perform a wired-AND operation with all PEs writing the bus. The wired-AND operation is useful for finding a global minimum value on the PEs.

C•RAM uses a write-enable register to conditionally enable individual PEs. Since a SIMD architecture requires each PE to execute identical instructions in lock-step, control structures, such as if-then-else, that execute conditionally require a mechanism for disabling individual PEs. Each PE in a C•RAM executes every instruction, but only the PEs with the write-enable register set will write the results back to memory. Writes to the X and Y registers are unconditional, as are network functions, so care must be taken in the design and use of control statements. Nested control structures can be implemented in software by maintaining a stack of write-enable bits in the PE's local memory.

Accelerix released a 4096-PE C•RAM-based processor in 1998. This Accelerix processor was used by Noah Aklilu in his M.Sc. thesis work as a research platform for developing a C•RAM controller [2].

### 2.2.5 Summary of PIM Architectures

The Terasys, IMAP and the early versions of C•RAM were all based on SRAM technology. SRAM has the benefits of fast access time, does not require refresh and can easily be implemented in a logic process. On the other hand, DRAM has the advantage of a much higher capacity since the memory cell consists of a single transistor and single capacitor whereas an SRAM cell is usually formed with 6 transistors.

The VIRAM-1 processor is more of a System-On-a-Chip (SOC) design. It uses embedded DRAM for both main memory and vector memory. A sequential MIPS processor is integrated into the same chip to issue instructions to the vector co-processor. Embedded DRAM allows much a much greater density than an SRAM-based design, again at the cost of higher latency. VIRAM amortorizes the increase in latency by organizing the memory into blocks with independent control logic so that multiple memory requests can be made in parallel. This also allows for multiple open rows which act as a small memory cache.

Later versions of C•RAM were designed for a DRAM process. C•RAM maps well into DRAM due to the simplicity of the PE. Since each PE is very small, it is easier to pitch-match to the narrow DRAM memory columns than with a larger processing element. As well, many metal layers are not required for routing since there is not as much logic to contend with. Even so, DRAM-based C•RAM implementations still require each PE to be pitch-matched to some small number (eg. 4) of memory columns rather than one.

Terasys and C•RAM are both bit-serial processors. This means that they operate on a single bit of data at each clock cycle. Since bit-serial processors require many clock cycles to compute a single value for integer numbers, they achieve great performance only through massive parallelism.

IMAP uses an 8-bit processor that consists primarily of a carry look-ahead adder. A shifter and look-up table is used to increase the performance of integer multiplies, although an add-shift method of multiplication is still performed. With a larger PE, IMAP will achieve higher performance for applications with moderate to high parallelism in comparison to a bit-serial PE like Terasys and C•RAM.

VIRAM is the most complex PE of all of the preceding architectures and maintains a 64-bit processor complete with floating-point operations. The complexity of the PE limits the number of processors in the VIRAM-1 chip to four. Like the SIMD extensions added to conventional microprocessors described in Section 2.1, VIRAM allows each register in the 64-bit processor to be treated as packed data, and operations on eight 8-bit numbers, four 16-bit numbers, two 32-bit numbers, or one 64-bit number are permitted. Unlike these designs, the width of the data is

Table 2.2: Comparison of Processor-In-Memory (PIM) Architectures

| | **Terasys** | **IMAP** | **VIRAM** | **C•RAM** |
|---|---|---|---|---|
| Memory Technology | SRAM | SRAM | Embedded DRAM | DRAM |
| Processor Size | 1-bit | 8-bit | Programmable | 1-bit |
| Operands | Memory | Registers | Registers | Memory |
| Integer Multiply | No | Partial[a] | Yes | No |
| Floating Point | No | No | Yes | No |
| Controller | External | External | Internal | External |

[a]IMAP uses a table lookup to improve multiplication performance

programmable through the Virtual Processor Width (VPW) register [21].

The major results of the PIM architecture survey are summarized in Table 2.2.

## 2.3 Image Processing Kernels

This research project focuses on evaluating the performance of the DSP-RAM architecture for key image processing applications. Image processing algorithms map well onto SIMD archictectures since it is common to execute the same sequence of instructions on all the pixels in an image. In order to achieve a quantitative measurement of image processing performance, we selected the following four common image processing operations:

1. Thresholding
2. Sobel dx and dy
3. Dilate and erode
4. Box and median filters

We will call these operators image processing kernels.

We assume that an image consists of a two-dimensional rectangular grid of 8-bit grey-scale pixel values. The case of a 24-bit colour image can be generalized from the grey-scale case because each colour pixel is composed of three independently processed 8-bit values. For the purposes of our study, we limit the image sizes to square images that range in size from 128 pixels per row to 4096 pixels per row.

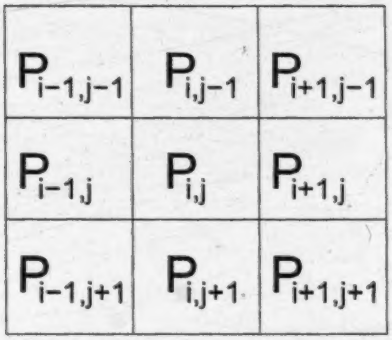



Figure 2.7: Interior pixel and neighbourhood

An *interior pixel* is a pixel that does not lie on one of the four image boundaries. The *neighbourhood* of an interior pixel is the set comprising the pixel and the set of eight immediately adjacent pixels going horizontally, vertically and diagonally. Figure 2.7 shows an interior pixel $P_{ij}$ and its neighbourhood.

### 2.3.1 Thresholding

Image thresholding is the simplest kernel in the study. The thresholding operation is accompished by replacing each pixel with a given fixed maximum value if the pixel value exceeds some given threshold; otherwise, the pixel is replaced with a given fixed minimum value. This implementation results in converting a 256-level grey-scale image into a 2-level binary image. The threshold kernel could be easily modified to use different threshold values for the maximum and minimum leaving unchanged the pixels that fall in between the two threshold values.

### 2.3.2 Sobel Dx and Dy

In most cases, the edge of an object appears in an image as a significant change in brightness, which may be spread out over several pixels [28]. By using first-order spatial derivatives in two orthogonal directions, the boundaries of the image can be approximated. The Sobel Dx and Dy operators uses the convolution masks shown in Equation 2.1.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.1}$$

(a) Original Image (b) Sobel Dx Image (c) Sobel Dy Image

Figure 2.8: Effects of applying Sobel gradients on an image

By using a convolution operation with the pixel neighbourhood and the mask, the dx and dy components can be computed, as shown in Equation 2.3.

$$dx = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} P(i-1,j-1) & P(i-1,j) & P(i-1,j+1) \\ P(i,j-1) & P(i,j) & P(i,j+1) \\ P(i+1,j-1) & P(i+1,j) & P(i+1,j+1) \end{bmatrix} \quad (2.2)$$

$$dy = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & -1 \end{bmatrix} \otimes \begin{bmatrix} P(i-1,j-1) & P(i-1,j) & P(i-1,j+1) \\ P(i,j-1) & P(i,j) & P(i,j+1) \\ P(i+1,j-1) & P(i+1,j) & P(i+1,j+1) \end{bmatrix} \quad (2.3)$$

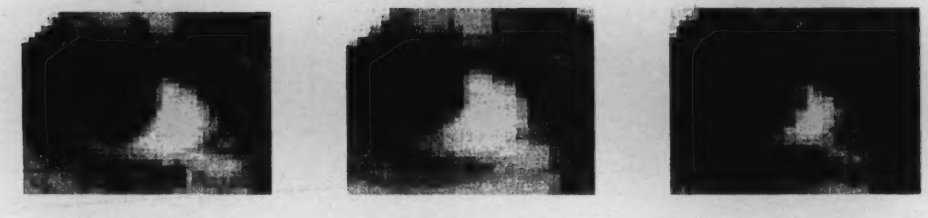
In order to detect an edge, the magnitude of the intensity must be computed as given by Equation 2.4.

$$Magnitude = \sqrt{dx^2 + dy^2} \quad (2.4)$$

These gradient operators are required for many different computer vision algorithms, including tracking and image recognition [16] [30]. The results of applying the Sobel gradient operators to the image of a square are shown in Figure 2.8. This illustrates that the Sobel Dx operator primarily detects changes in the horizontal direction—yielding vertical lines, whereas the Dy operator detects changes in the vertical direction—resulting in horizontal lines.

### 2.3.3 Dilate and Erode

Image dilation and erosion are classified as image morphological operations [28]. Image dilation replaces the pixel value with the maximum pixel value in the original neighbourhood of pixels, while erosion replaces the pixel value with the minimum of the original neighbourhood. Figure 2.9 shows the results of applying the dilate



(a) Original Image (b) Dilated Image (c) Eroded Image

Figure 2.9: Effects of dilate and erode on an image

and erode operators to an image. For grey-scale images, dilation has the effect of brightening the image, as well as growing the bright regions of the image. Erosion has the opposite effect. Since it uses the minimum pixel value, the entire image will appear darker, and the dark regions of the image will grow. These effects are evident in Figures 2.9(b) and 2.9(c). Dilation and erosion are the most primitive of the morphological operators and are used to build more complex morphological operations such as image open and close. Morphological operators are useful in pattern matching applications and in image analysis [30].

### 2.3.4 Box and Median Filters

The box filter is used to remove noise from an image prior to processing. This process is called image smoothing. The box filter replaces the pixel with the average of all the pixels in a neighbourhood. While this smoothing process makes the image cleaner by removing random noise, the processed image is not as sharp as the original because fine image features are averaged away along with the noise. When the pixel being processed is on a boundary in the image, the neighbourhood contains pixel values from two or more intensity regions. This blurs the boundary in the processed image.

The median filter is a popular alternative to reduce this blurring [30]. The median filter computes the median value of the pixels in the neighbourhood and uses this as the new pixel value. By sorting the group of pixels, the median value can easily be determined. For implementation purposes, the entire neighbourhood does not have to be sorted. Instead, the sorting algorithm is modified so that once the

(a) Original image (b) Box filtered image (c) Median filtered image

Figure 2.10: Effects of noise filtering on an image

median value has been found, the algorithm stops. While the median filter does a better job at smoothing an input image, it generally requires more computation time than a box filter [30]. As shown in Figures 2.10(b) and 2.10(c), both filter algorithms have fairly similar results when smoothing the image.

## 2.4 Low-Power VLSI Design

In order to describe how DSP-RAM achieves low-power operation, it is useful to review the major components of power dissipation in digital CMOS circuits. Total power consumption is composed of a dynamic power consumption term and a static power consumption term, as given by Equation 2.5.

$$P_{total} = P_{static} + P_{dynamic} \tag{2.5}$$

### 2.4.1 Charging and Discharging Capacitance

Charging and discharging capacitance is the dominant form of power dissipation in digital circuits [44]. This source of power dissipation is categorized as dynamic power dissipation since it is caused by the switching activities of the circuits. The power resulting from charging and discharging capacitance is expressed by Equation 2.6.

$$P = C_{total} V^2 f \tag{2.6}$$

The static CMOS inverter in Figure 2.11 will be used to illustrate the dynamic power dissipation of a gate. When the input signal is at logic level '0' (GND), the PMOS

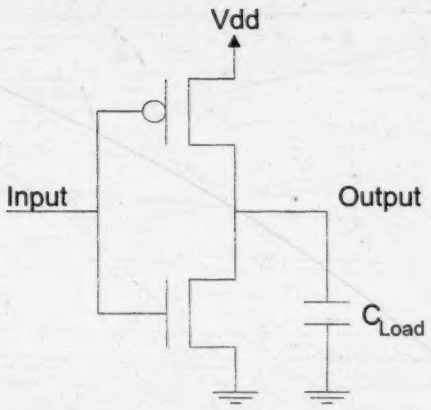



Figure 2.11: Static CMOS inverter

transistor is turned on and the NMOS transistor is turned off. This causes current to flow through the PMOS transistor and charge the load capacitance. When the input signal changes from a logic '0' to a logic '1', the PMOS transistor turns off, and the NMOS transistor turns on. This causes the energy stored in the capacitor to be discharged through the NMOS transistor. As the frequency of the input switching increases, the amount of energy required from the power supply increases in direct proportion.

### 2.4.2 Short Circuit Current

The second form of dynamic power consumption is from short circuit current (also called crowbar current). When a static CMOS gate, like the inverter shown in Figure 2.11, is in the process of switching, there is a direct path from $V_{DD}$ to $GND$. This is caused by the non-zero rise or fall time of the input signal. When the input signal momentarily lies between $V_{T_N}$ and $V_{DD} - V_{T_P}$ both the PMOS and NMOS transistors will be on. This will cause current to be drawn and the energy will be dissipated as heat through the transistors, as shown in Figure 2.12. The inverter used in this figure was simulated in TSMC 0.18-$\mu$m technology with an NMOS width of 1 $\mu$m and a PMOS width of 2 $\mu$m. The input signal has a rise and fall time of 250 picoseconds. These parameters cause a short circuit current of 90 $\mu$A when the input transitions from low to high, and 135 $\mu$A when the input transitions from

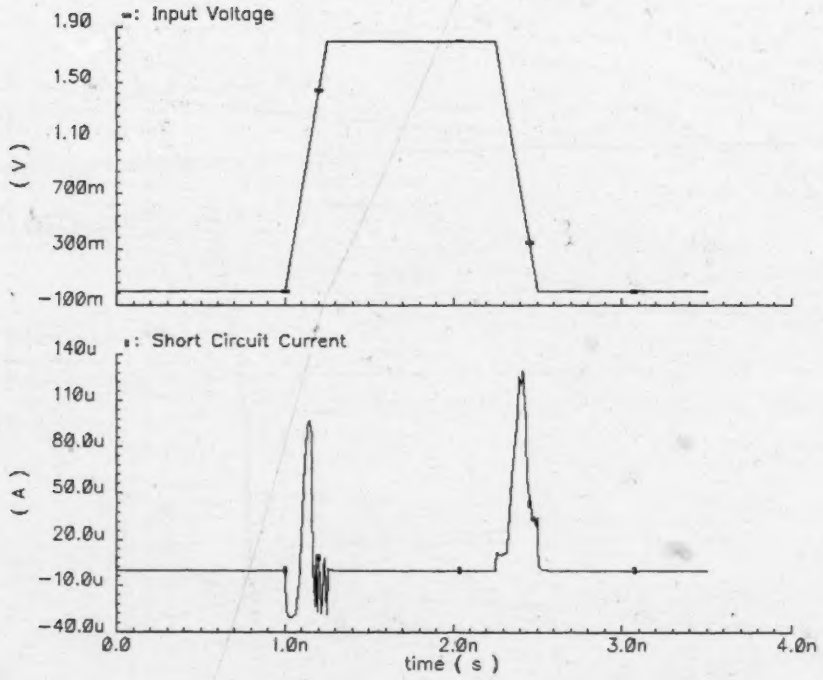



Figure 2.12: Short circuit current in a static CMOS inverter

high to low. The variation in the short circuit current is caused by the differences in the PMOS and NMOS transistors.

### 2.4.3 Leakage Current

Leakage current exists in every transistor and is generally not desired for normal device operation. Leakage current is a portion of the static power dissipation term in Equation 2.5. The first form of leakage current occurs when the drain of an NMOS transistor is at $V_{DD}$ or the drain of a PMOS transistor is at $GND$. This is called a Reverse Biased PN-junction. The PN junction is formed from the drain terminal to the substrate for NMOS (n-well for PMOS). The magnitude of this current depends on temperature, area and the fabrication process. In any event, this leakage current is typically of the order of $1\text{pA}/\mu m^2$ and is usually insignificant in comparison to the dynamic power [44].

The second form of leakage current is from the subthreshold channel leakage. This subthreshold current occurs when the gate voltage is below the threshold voltage (ie. the transistor is off). The subthreshold conduction current is expressed by Equation 2.7 [44].

$$I_{sub} = I_0 e^{(V_{gs}-V_t)/(\alpha V_{th})} \tag{2.7}$$

In this equation, $I_{sub}$ depends on process fabrication characteristics and temperature (the $\alpha$ and the $V_{th}$ parameters, respectively), the gate voltage $V_{gs}$, and the threshold voltage $V_t$. When the transistor is completely off, the gate voltage is much less than the threshold voltage so $V_{gs}$ can be approximated by 0. As the threshold voltage decreases, which is necessary as the operating voltage of the circuit decreases [25], the subthreshold current increases due to the negative exponent term. This leakage current is estimated to be increasing by $7.5\times$ for each technology generation [8].

### 2.4.4 Static Power Dissipation

Aside from leakage currents, properly designed digital circuits in static CMOS do not draw static current. All non-leakage current should only occur during switching through the charging and discharging of capacitance and short-circuit current.

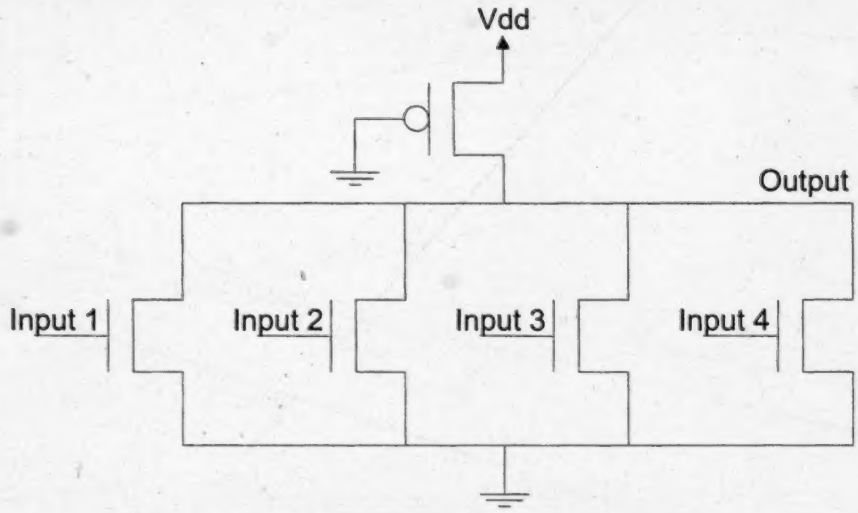



Figure 2.13: Pseudo NMOS 4-input NOR gate

There are times, however, that deviation from purely static CMOS design is required. Consider the pseudo NMOS circuit in Figure 2.13. The benefit of this design is that it eliminates the series connection of four large PMOS transistors. The disadvantage is that whenever one of the input signals is active, there is static current from $V_{DD}$ to $GND$ because the PMOS transistor is always turned on. However, there is no static current drawn when all the input signals are at logic level '0'. Therefore, this type of circuit is useful when the probability that the input signals will activate the gate is extremely low, such as in the system reset or system test circuitry [44].

### 2.4.5 Low-power DSP-RAM Architecture

Now that the primary sources of power dissipation in digital CMOS circuits have been reviewed, low-power features of the DSP-RAM architecture can be discussed. These low-power characteristics are common among all parallel PIM-style architectures [21] [13]. The dynamic power dissipated through charging and discharging capacitance, expressed in Equation 2.6, is usually the primary component of total power consumption. Parallel processing in memory offers opportunities to reduce both the operating voltage and clock frequency, which decreases the total power

consumed by the system.

The first method of power reduction is to reduce the frequency at which the off-chip bus capacitance is driven. The largest capacitance values are those found at the pin drivers since they must drive the capacitance of the printed circuit board (PCB) traces, which can be in the 50-100 pF range [38]. By making use of the high internal bandwidth of the memory core and by processing the data inside the memory chip, the data does not have to be driven off-chip across the high-capacitances wires. This minimizes the activity on the off-chip data bus and can save substantial power. For image processing applications that require large quantities of data to be passed from the memory to the CPU for processing, significant power savings can be achieved by processing the data inside the memory chip.

The second method of reducing the power originates from the parallelism. Parallel datapaths allow the clock frequency of a processor to be reduced and still maintain the same throughput of a single datapath operating at a much higher frequency. However, this reduced frequency is not directly responsible for lower power operation. In order to maintain a constant throughput, multiple datapaths are required which increases the total capacitance—offsetting the reduced frequency. In fact, the increase in capacitance is generally greater than the decrease in operating frequency. This is due to the interconnect between multiple processing elements, as well as the overhead of controlling them.

The lower operating frequency that is permitted by a parallel system, while causing a proportional increase in capacitance, allows for a reduced operating voltage. Since the dynamic power is proportional to the square of the operating voltage, this provides significant power savings. Equation 2.8 shows that the maximum frequency is dictated by $V_{DD}$ [9] [18] [44].

$$f_{max} \alpha (1 - \frac{V_t}{V_{DD}})(V_{DD} - V_t) \tag{2.8}$$

By establishing that the parallel system will operate at a lower clock frequency than a uniprocessor system, a lower operating voltage can be used. Figure 2.14 shows the operating voltage required for a CMOS inverter in TSMC 0.18$\mu$m technology at varying clock frequencies. This figure shows that an inverter operating at 1.8 V $V_{DD}$

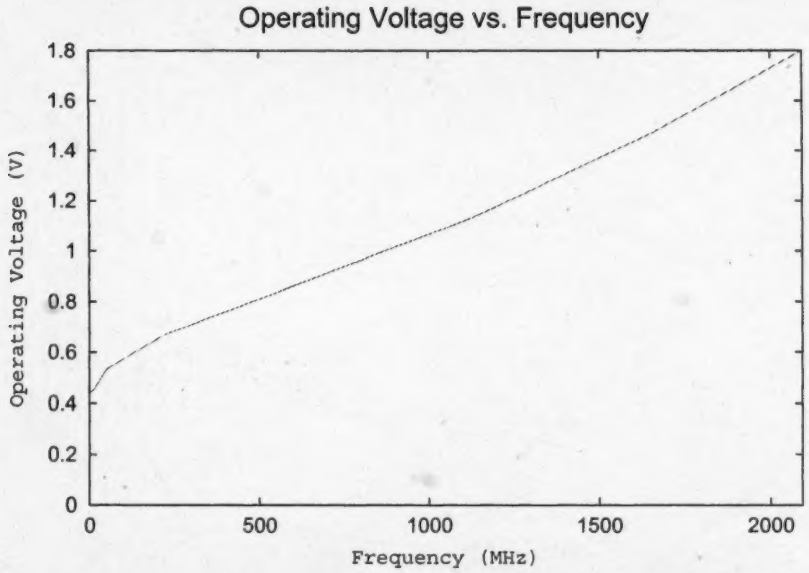



Figure 2.14: Minimum required operating voltage versus frequency

can be switched at a rate of just over 2 GHz. By allowing the inverter to operate at approximately 1 GHz, the operating voltage may be reduced to 1 V and the circuit will still perform correctly. Considering only the reduced operating voltage, the power will be reduced by 69%.

Figure 2.15 shows two equivalent systems—a uniprocessor system and a parallel system. Equation 2.9 represents the dynamic power dissipation of the unipro-

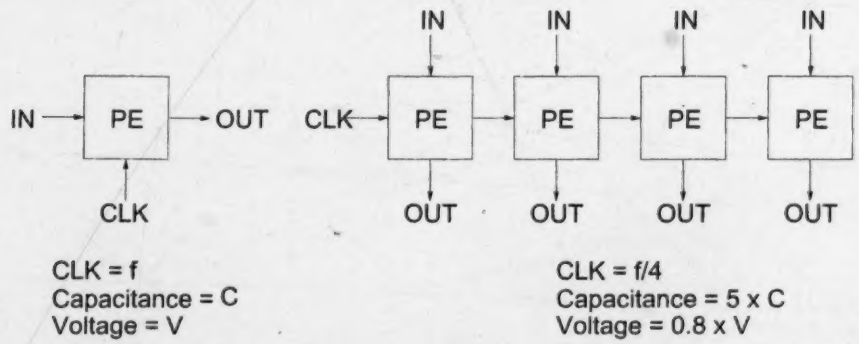



Figure 2.15: Power savings in parallel processor systems

cessor system in Figure 2.15 (left side).

$$P_{uniprocessor} = CV^2 f \tag{2.9}$$

The same can be done for the parallel system in Figure 2.15 (right side).

$$P_{parallel} = (5C)(0.7V)^2(\frac{f}{4}) = 0.6125\, P_{uniprocessor} \tag{2.10}$$

As shown in Equations 2.9 and 2.10, the power savings of a parallel system can be quite substantial by allowing for a lower operating voltage.

Additional low-power design techniques, such as reduced-swing buses between each processing element and circuit gating for registers and memory cells [24], could be used to further reduce the total system power consumption in the DSP-RAM architecture.

## 2.5 Development Platform

Our research architectures were implemented on the Rapid-Prototyping Platform (RPP) that was lent from the Canadian Microelectronics Corporation (CMC). The RPP uses the ARM® Integrator™ series products and consists of an Integrator/AP module, the system motherboard; an Integrator/CM7TDMI core module, the microprocessor; and the Integrator/LM-XCV600E+ logic module, and the FPGA. Figure 2.16 shows a picture of the development platform. This system was put together by ARM to provide a platform for prototyping SOC designs before committing the design to silicon. The platform also supports hardware/software co-development. The three modules communicate using the ARM Microcontroller Bus Architecture (AMBA) bus protocol [4]. The core module and motherboard have previously programmed FPGAs that implement the AMBA protocol. The logic module, however, must have an AMBA controller synthesized into the design in order to communicate over the system bus.

Today's FPGAs offer an excellent prototyping platform for parallel PIM-style architectures. FPGAs are inherently parallel. They are composed of an array of identical configurable logic blocks (CLBs). High-end FPGAs offer internal RAM and configurable multiplier cells. For example, the Xilinx Virtex-2 pro series

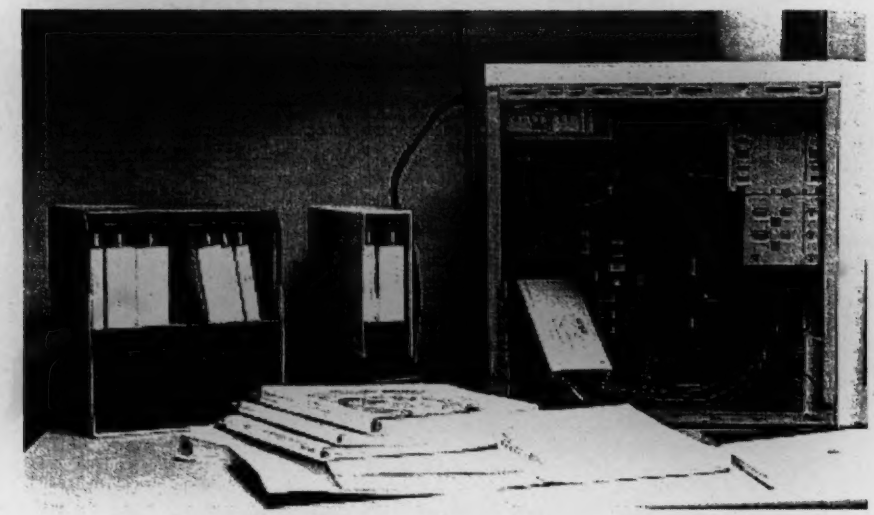


Figure 2.16: ARM Integrator series rapid prototyping platform [10]

XC2VP125 FPGA, offers 10,008 Kb of memory, 556 18x18 multiplier blocks and 13904 CLBs [40]. This particular FPGA also contains four PowerPC sequential cores. One of these microprocessors could be used as a controller for the SIMD array, while the remaining ones could be used for sequential processing. As future generations of FPGAs are released, the number of logic gates and the amount of memory will increase. A SIMD design will be able to immediately benefit from these increases by simply resynthesizing the design with more processing elements. Currently a 32-PE DSP-RAM system can be implemented in one Virtex 2000E FPGA, with each bank of memory containing 256 16-bit words. We estimate that a 64-PE DSP-RAM system, with each memory bank containing more than 2K 16-bit words, could be synthesized into one XC2VP125 FPGA

### 2.5.1 Integrator/AP

The Integrator/AP module [7] is the system motherboard. The block diagram for the Integrator/AP board is shown in Figure 2.17. The core of this module is the system controller FPGA. The system controller is used to interface to the logic and core modules in the system. It is also used as the system bus arbiter and interrupt controller. The system controller also maintains three counter/timers as well as

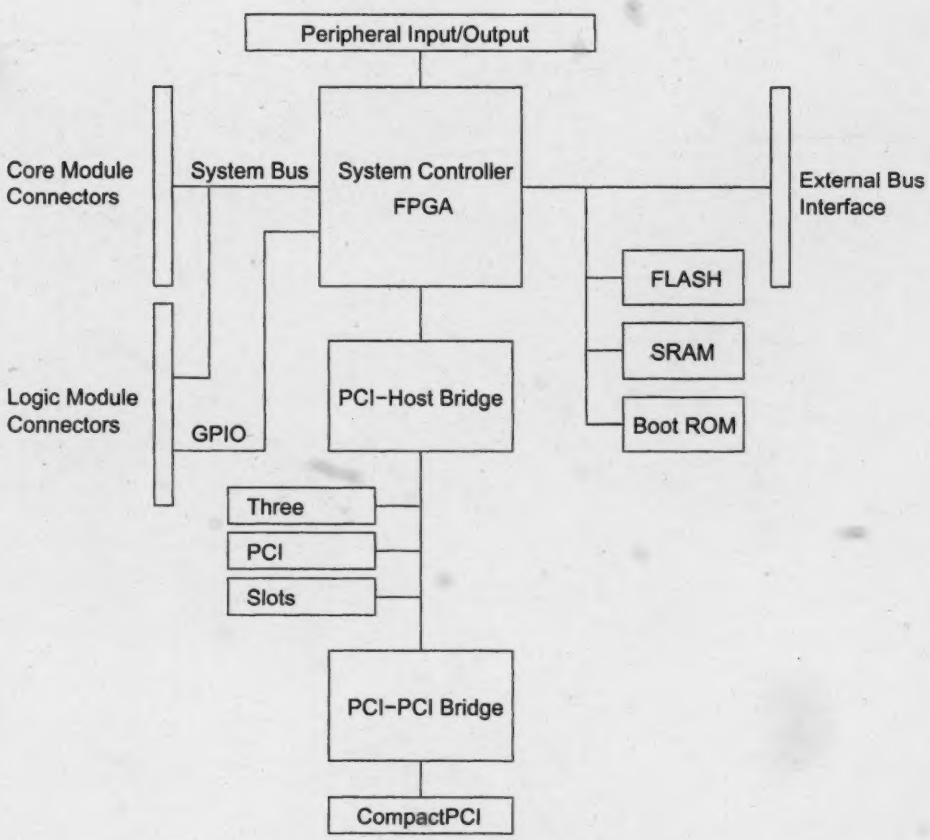



Figure 2.17: Integrator/AP motherboard [7]

system status and control registers.

The Integrator/AP also has a PCI host bridge which is used to connect three PCI slots to the AMBA bus that is used in the rest of the system. The PCI slots allow the system to be expanded depending on the application. There is also a PCI/PCI bridge which is used to map the CompactPCI slot onto the system bus.

The motherboard provides 32 MB of flash memory and 512 KB SSRAM which is accessible to the logic and core modules through the expansion bus interface. A total of five logic and core modules can be stacked onto one Integrator/AP. However, only four logic modules or four core modules can be stacked together. There is also an external bus interface and a peripheral input/output header for system expansion.

For system input/output (I/O), the Integrator/AP has a keyboard and mouse interface (KMI), two UART serial ports, four general-purpose LEDs and two alphanumeric displays. These are all accessible via the system controller from either a logic or core module.

### 2.5.2 Integrator/CM7TDMI

The Integrator/CM7TDMI module [5] is the main controller in the system. It is based on an ARM7TDMI microprocessor core, as shown in Figure 2.18. The core module has a single SDRAM DIMM slot that supports up to 256 MB of SDRAM memory. When multiple core modules are stacked together, each core module has access to the other board's SDRAM. The FPGA on the core module is used as a system bus bridge and acts as an interface between the core module and the motherboard, the local SDRAM and, optionally, the other core modules. The ARM7TDMI core module has a programmable CPU clock, up to a maximum of 160 MHz, and a programmable memory bus clock, up to a maximum of 66 MHz.

### 2.5.3 Integrator/LM-XCV600E+

The Integrator/LM-XCV600E+ module [6] is the FPGA logic module for the system. The logic module used in this research is the Xilinx Virtex 2000E FPGA. This Virtex 2000E has 9600 CLBs and 640 Kbits of internal memory [41]. Figure 2.19 shows the block diagram of the logic module. The FPGA has access to two pro-

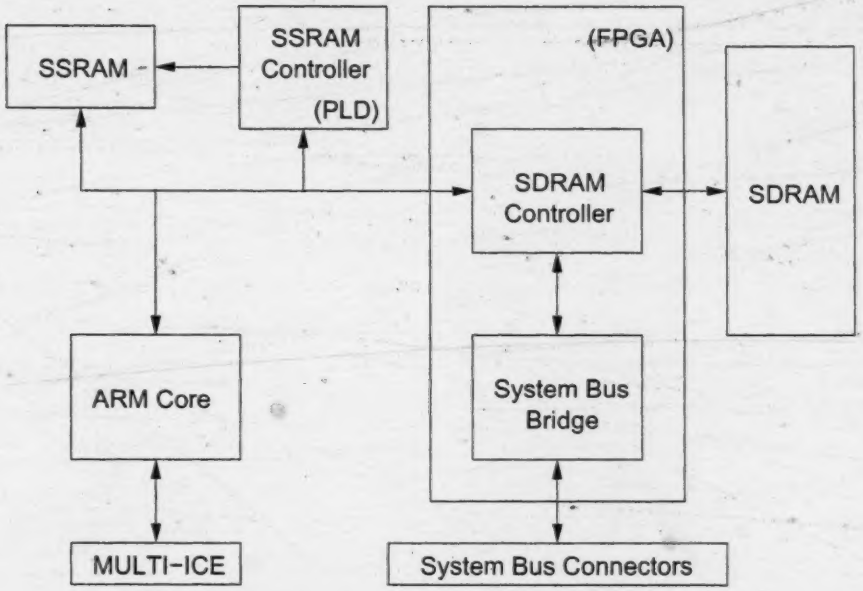



Figure 2.18: Integrator/CM7TDMI microprocessor core module [5]

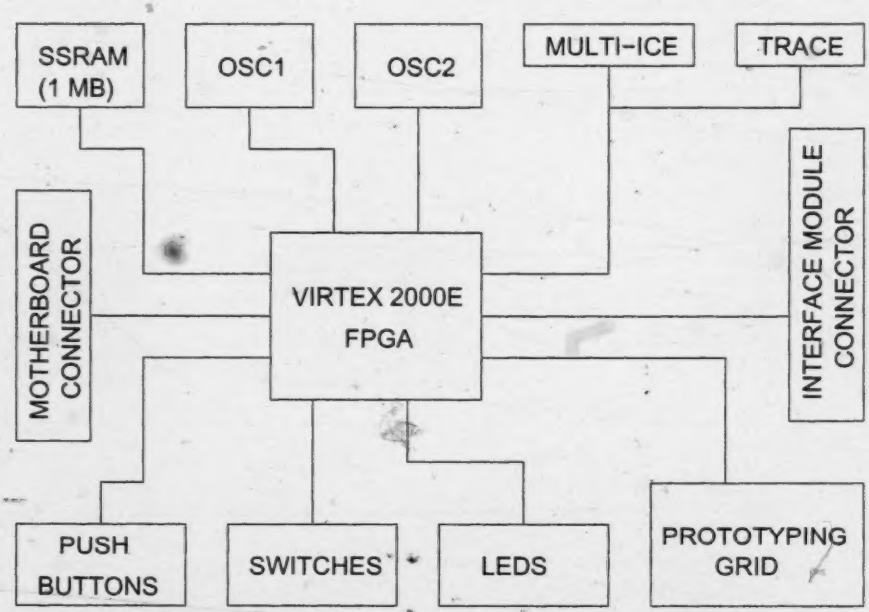



Figure 2.19: Integrator/LM-XCV600E+ logic module [6]

grammable oscillators to control the clock frequency of the design. There is also 1 MB of external SSRAM on the board. Also provided are a push-button, switches and LEDs that can be configured for system input and output. The logic module also has direct access to the external interface module connector. There is a configuration PLD and flash memory that are not shown in the block diagram. These chips are used to store FPGA configurations and are loaded into the FPGA on reset.

# Chapter 3

# Evaluation of the Original DSP-RAM Architecture

DSP-RAM is a PIM-style architecture based on C•RAM. As described in the previous chapter, C•RAM uses simple bit-serial processors to exploit the massive parallelism available in some applications. While these straightforward processors provide many advantages, they also suffer from certain disadvantages. The first disadvantage is that an application must exhibit massive parallelism (ten thousand to one million processors) in order for the performance improvements to be realized. This becomes an issue for applications that exhibit only moderate parallelism (fifty to one thousand processors) [2]. The second disadvantage is that multiplication is a very high latency operation on a bit-serial processor. This implies that applications requiring many multiplications, such as matrix multiplication, will be a poor fit for C•RAM.

DSP-RAM solves these problems by introducing a more sophisticated PE that is built around a 16-bit multiply-accumulate (MAC) unit. A larger, more complicated PE results in a smaller degree of implementable parallelism, typically ranging from 64 to 256 PEs. This allows DSP-RAM to be easily mapped to alogithms that show moderate levels of parallelism. Further, a pipelined MAC gives DSP-RAM the ability to perform MACs at an averaged rate of close to one cycle per operation.

Using both the DSP-RAM emulator and the C•RAM compiler [13], we can compare the performance of both vector addition and vector multiplication on the different platforms. Figure 3.1 compares the performance of vector addition for

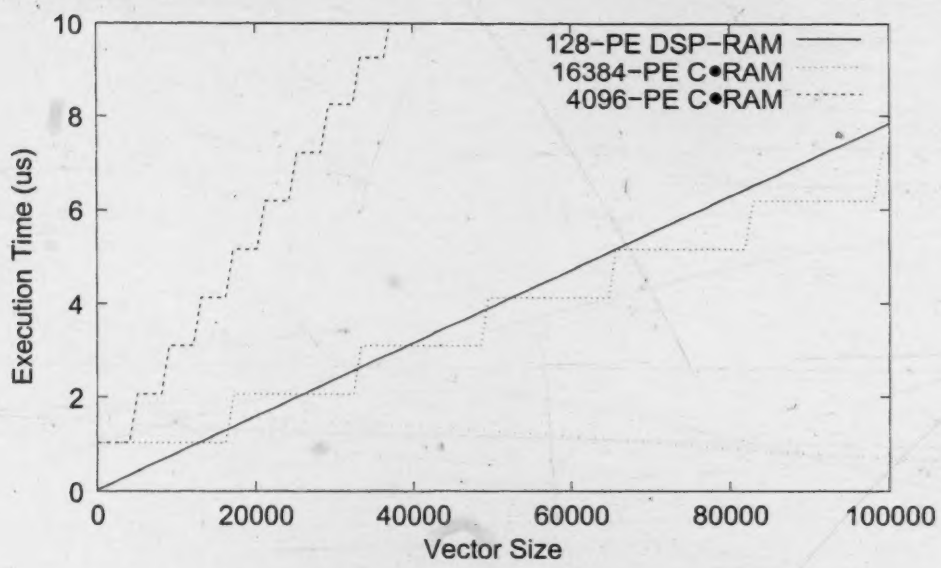



Figure 3.1: SIMD 16-bit addition performance

a 128-PE DSP-RAM, a 4,096-PE C•RAM and a 16,384-PE C•RAM. The graph shows that the 4,096-PE C•RAM can never match the performance of the 128-PE DSP-RAM. A 16,384-PE C•RAM will match the performance of DSP-RAM and eventually surpass it as the vector size increases. As expected, DSP-RAM is very good at moderately-parallel applications, whereas C•RAM excels at massively-parallel applications.

Figure 3.2 shows a similar plot for multiplication. Unlike addition, where performance benefits were realized at vector lengths of around 12,000 for C•RAM, multiplication requires a vector length of around 250,000 in order to match the performance of a 128-PE DSP-RAM. As well, the 16,384-PE C•RAM that achieved performance improvements over DSP-RAM for addition, never out-performs DSP-RAM on multiplication.

While we have been focusing on the disadvantages of C•RAM and how DSP-RAM has been designed to overcome the limitations of C•RAM, it is important to also discuss the areas where C•RAM retains an advantage over DSP-RAM. The size and complexity of the DSP-RAM processing element reduces the amount of memory that can be included in the chip. Therefore, memory-intensive applications

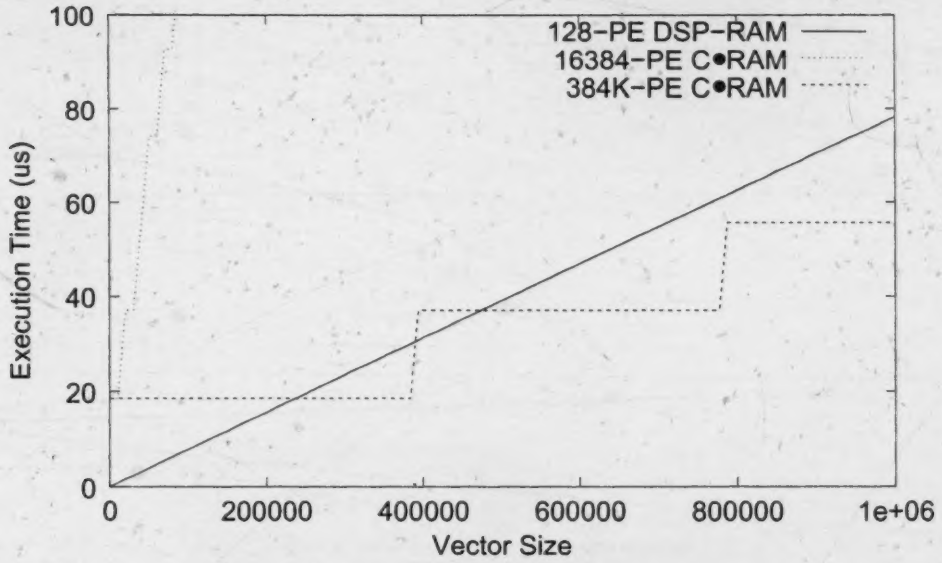



Figure 3.2: SIMD 16-bit multiplication performance

are better suited for C•RAM. C•RAM also supports variable-length operands, unlike DSP-RAM which is restricted to 8 or 16-bit fixed-point data values. Further, due to the bit-serial nature of C•RAM, as the number of bits in the operands is decreased, the performance of C•RAM increases. In summary, the DSP-RAM architecture targets multiplication-intensive applications that exhibit moderate levels of parallelism, and that do not require large amounts of memory for each processor.

## 3.1 DSP-RAM Architecture

The original DSP-RAM architecture, which existed before this thesis project, is shown in Figure 3.3. It is a special-purpose register architecture [17] that can use memory or register operands as inputs to the one ALU. Data is stored locally in two 16-bit memory banks (SRAM A and SRAM B). Temporary storage is available in three accumulator registers (ACC 2, 1, and 0) as well as a broadcast bus register (BBREG), the shift register (SHIFT REG) and several pipeline registers within the multiplier (not shown). Data can be transferred into a PE in one of three ways: (1) over a 16-bit global broadcast bus; (2) from the left or right nearest-neighbouring PEs over a 48-bit left/right shift bus; and (3) by write operations initiated by the

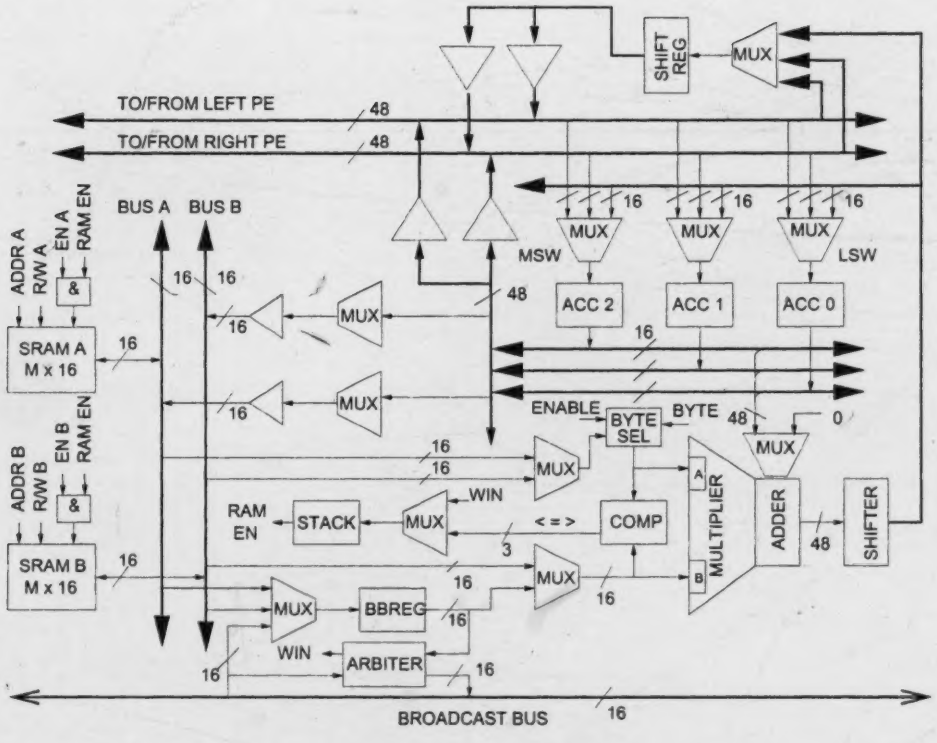



Figure 3.3: DSP-RAM processing element

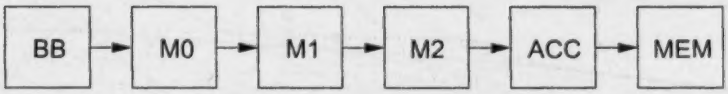



Figure 3.4: Broadcast bus to memory pipeline

external controller or CPU into 16-bit memory locations in the memory banks. The sub-blocks of the PE will be described in detail in the following sub-sections.

### 3.1.1 Networking

When the same constant needs to be stored into each PE, the broadcast bus is used to broadcast the value from an assumed controller into each PE's broadcast bus register (BBREG). The constant may then be stored into the local memory banks, or remain in the BBREG. Storing a single constant to a local memory location requires six cycles, as shown in Figure 3.4. The first cycle (BB) broadcasts the constant on the broadcast bus and loads it into the broadcast bus register. The next three cycles (M0, M1, and M2) are the pipeline stages inside the multiplier. M0 latches the multiplier operands from the BBREG and the constant '1' stored in a memory bank. M1 is a pipeline register in the middle of the multiplier, and M2 is a pipeline register at the end of the multiplier and before the adder. The fifth cycle (ACC) loads the data into 16-bit accumulator register 0. The sixth and final cycle loads accumulator 0 into memory. An array of constants can be stored in memory very easily by pipelining the process of loading a single constant. The total number of cycles required then depends on the length of the array as shown in Equation 3.1.

$$N_{CYCLES} = 5 + N_{CONSTANTS} \tag{3.1}$$

As the number of constants increases, the number of cycles required per constant asymptotically approaches one.

The left/right shift bus can also be used to load data into PE local memory. The only stipulation is that the data must be loaded into the same location for each PE because DSP-RAM does not support autonomous addressing (each PE must access its local memory at the same memory address). Data moves through the processor array via the left/right shift bus, using the accumulator or shift register as

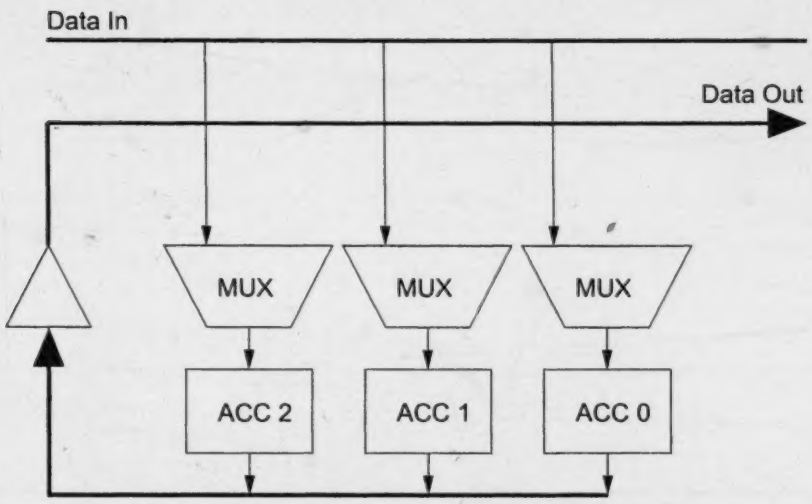



Figure 3.5: Data path used to shift data through the PE array

temporary storage, as shown in Figure 3.5. Once all the data values are loaded into the accumulator, one memory write is used to store all the data values in parallel in all PEs. The total number of cycles, $N_{CYCLES}$, to load $N_{DATA}$ values is given by Equation 3.2.

$$N_{CYCLES} = 3 + \frac{N_{Data}}{3} \tag{3.2}$$

Three 16-bit data values can be shifted onto the 48-bit shift bus at each cycle. Therefore, as the number of data values increases, the number of cycles per data value approaches one-third. This method also only requires three memory accesses to write all the data values. Since the memory is not accessed during the shift cycles significant power savings can be realized over a sequential memory load that requires a memory access for each data value. Using this method, two separate performance improvements can be realized. If the total time required to load a block of data is critical, data values can be sequentially loaded in parallel with shifting data—resulting in approximately one-quarter cycle for each data value. If processing time is critical, the shift register can be used instead of the accumulator, and the PE can process data through the ALU in parallel during the shifting of data. Since the memory banks are being used in each of these situations, both techniques cannot be used simultaneously.

The final option for loading data into DSP-RAM is by simply accessing DSP-RAM as a conventional 16-bit SRAM. Storing data into the memory banks is done sequentially through the controller via a single 16-bit global data port. This method requires one cycle for each 16-bit data value to be addressed separately and stored and hence does not utilize the full memory bandwidth available at the sense amplifiers. However, in some circumstances, data is received in a format that does not map efficiently onto the SIMD array of processors so the shift bus cannot be directly used. For example, in Figure 3.6 four data elements are received for each PE at a time. The data then has to be reformatted into groups of three data values in order to be shifted in through the shift bus. This regrouping causes the data to be misaligned on the PEs and thus the data will be stored out of order when each PE stores the accumulator values into local memory. By writing directly to local memory, the data values are stored in the correct location without having to perform any regrouping. This method is also useful when the data originates from a device that supports direct memory access (DMA). In this case, the device can DMA directly into the DSP-RAM through a separate DMA controller.

### 3.1.2 ALU

The core processing unit is a pipelined multiplier-accumulator (MAC) that multiplies two 16-bit operands and adds the resulting 32-bit product via a shifter into a 48-bit accumulator register. The operand for the A input of the multiplier can come from memory bank A, memory bank B, or any of the three accumulator registers. The operand for the B input of the multiplier can come from memory bank B, the BBREG, or any of the three accumulator registers. The BBREG can be loaded from the broadcast bus, memory bank A, memory bank B, or any of the three accumulator registers.

The value stored in the accumulator can be optionally added to the product from the multiplier. The accumulator maintains a generous 16 guard bits to protect against overflow. After the optional addition stage, the result passes through the shifter. This shifter can perform logical shift left (LSL), logical shift right (LSR), arithmetic shift right (ASR) or no shift. In order to keep the area of the shifter down,

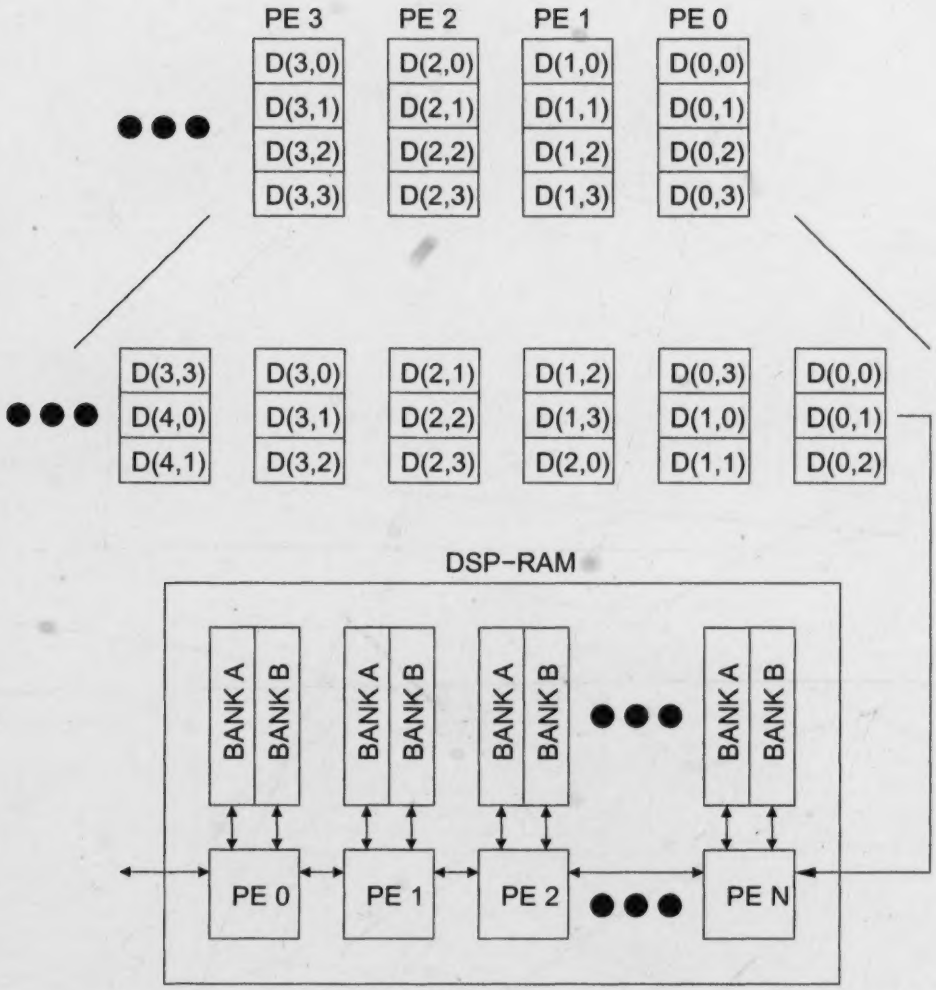



Figure 3.6: Data stored incorrectly through the shift bus

Figure 3.7: ALU pipeline

the number of bits to shift the result is constrained to four or eight different values. The final result can be stored into the accumulator or into the shift register.

The ALU is in the path of the same pipeline that is used to pass data from the broadcast bus to memory, as shown in Figure 3.4. If the BBREG is not used as an operand, the BB stage of the pipeline is not required, as shown in Figure 3.7. In this pipeline, if both operands are from the memory banks, there is a potential structural hazard. A structural hazard is caused when a single resource is required by two separate sources during over-lapping pipeline stages. This hazard exists in DSP-RAM when both the M0 and MEM stages access the same memory bank. The M0 stage is used to fetch the operands from memory, while the MEM stage is used to write the accumulator registers to memory. If the M0 stage and MEM stage were to attempt to access the same memory bank in the same clock cycle, resource contention would occur. In order to avoid this, a stall must be inserted into the pipeline—which results in a wasted cycle.

### 3.1.3 Conditional Operation

A comparator and hardware stack are provided to support conditional branching in the form of an if-then-else control structure. First a condition (selected from among the $<$, $=$ or $>$ signals from the comparator, or the MIN signal from the arbiter) is evaluated in all PEs in parallel and then pushed into the topmost location of the hardware stack. The local RAM enable signal (RAM EN) is formed by taking the logical AND of the occupied locations in the stack. For those PEs for which RAM EN is true, write operations to the RAM banks and the registers will be enabled; the other PEs will be disabled from changing the contents of their RAM banks and registers, thus preserving their state. To switch between the then-part and the else-part of the inner-most if-then-else construct, the enable signal at the top of the stack is toggled. When the end of the else-part is reached, the hardware stack is popped.

In this way, nested if-then-else statements can be supported (up to some hardware-determined maximum nesting depth). While performing conditional operations on SIMD computers, some processors will effectively be disabled. This decreases the benefits of parallel execution, and so conditional instructions should be used sparingly.

### 3.1.4 Arbiter

An arbiter circuit is provided in each PE to allow the DSP-RAM to perform a global minimum comparison operation over the wire-ANDed broadcast bus. This comparison is required at the end of some calculations to determine which PE has obtained the smallest result. To simplify the design of the arbiter, the circuit treats the contents of the broadcast bus as an unsigned integer. At first, all PEs will drive the contents of the BBREG onto the broadcast bus. The arbiter will then compare the value on the broadcast bus and the BBREG, starting with the most significant bit and ending with the least significant bit. If the bits do not match, then the arbiter has determined that another PE has a smaller value than its own and the WIN signal is set low and the PE no longer broadcasts the contents of the BBREG. It is important to realize that for a wired-AND bus the only time the bits will not match is when the locally supplied bit is a '1'. At the end of this process, the PEs with the WIN signal still set high have the same smallest value. In order to keep this process synchronous, the arbiter is implemented as a 17-cycle operation. The first clock cycle enables the arbiter and drives the contents of the BBREG onto the broadcast bus. The next sixteen cycles compare one bit of the broadcast bus and the BBREG, starting with the most significant bit (MSB). If a mismatch is found, the arbiter lowers the WIN signal and stops driving the broadcast bus.

## 3.2 The C++ Emulator

The recommended design flow for algorithm development on DSP-RAM is shown in Figure 3.8. Algorithms are first developed and verified for a sequential machine. This provides output vectors and timing to compare against the parallel

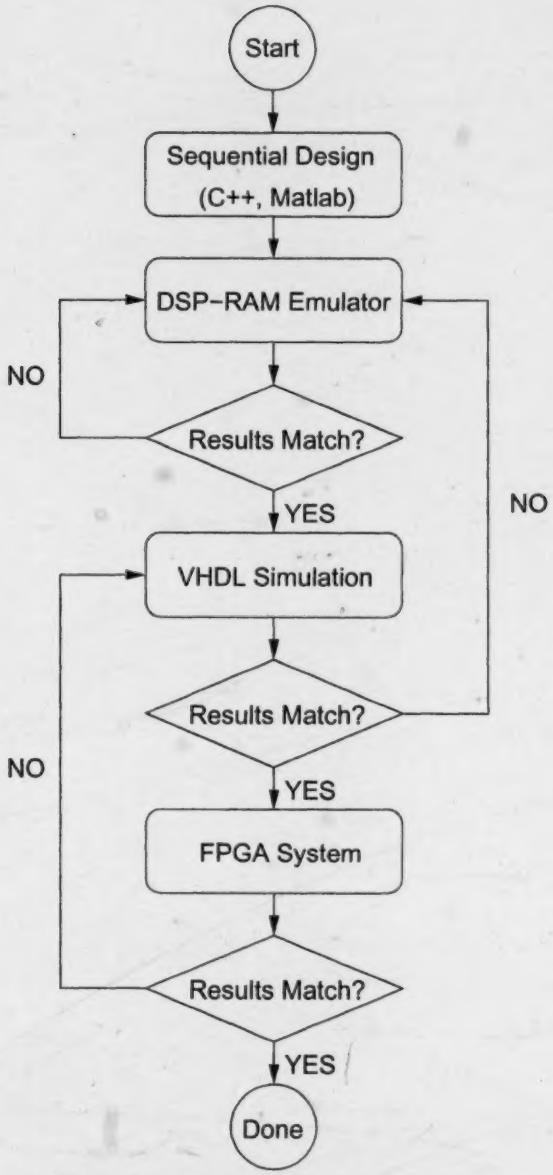



Figure 3.8: DSP-RAM design flow

implementation. Next, parallel implementations of the algorithms are developed on the DSP-RAM emulator. This C++ emulator provides accurate cycle counts for the algorithm. The emulator allows a user to debug the algorithm in a controlled software environment. It is written in C++ so DSP-RAM programmers are free to use any C++ design environment they are familiar with. Useful features provided through the software emulator are:

- Single-stepping through code
- Printing state information
- Setting break-points
- Outputing cycle counts

Once a working version of an algorithm has been coded on the emulator, the emulator will report the number of cycles required on DSP-RAM to execute the code. This number can be converted into actual time by choosing a clock frequency for the DSP-RAM implementation. This execution time can then be compared with that obtained on a conventional workstation. At this point, the developer is free to analyze the bottlenecks for the design and to try to speed up the performance by implementing portions of the algorithm differently.

Once the design has been completed on the emulator, test vectors are created for the algorithm and used as the stimulus for the VHDL model. This step is used to find potential problems in both the emulator model and the VHDL model. After the algorithm is verified through VHDL simulation, it is implemented in the real FPGA system and compared against the expected results. Full source code for the emulator is included in Appendix B.

## 3.3 VHDL Implementation

The DSP-RAM architecture was modeled in a synthesizable subset of VHDL. The model was synthesized using the Synopsys' FPGA Compiler 2 software for FPGA implementations and with Design Compiler for 0.18$\mu$m standard cell implementations.

The architecture in Figure 3.3 contains seven components that are described and analyzed in more detail:

1. Multiplier
2. Adder
3. Comparator
4. Memory
5. Shifter
6. Stack
7. Arbiter

The multiplier, adder, and comparator were instantiated as Synopsys DesignWare components [34]. Developing with DesignWare components abstracts away the details of the implementation from our design, but does not constrain the design to a specific target. The DesignWare libraries can be synthesized for all the FPGAs supported by the FPGA Compiler as well as for the $0.18\mu m$ standard cells. The memory component was instantiated as a BlockRAM [41] model for the Xilinx FPGA implementations. For an ASIC implementation, SRAM characteristics from the TSMC foundry for the $0.18\mu m$ process were used.

### 3.3.1 Shifter

The shifter is capable of providing a logical shift left (LSL), a logical shift right (LSR), an arithmetic shift right (ASR) or no shift operation. An LSR is performed by masking the sign extension bit and using the ASR unit. When an ASR is selected, the most significant bit of the input is shifted into the ASR unit, otherwise a logic '0' is shifted in. Figure 3.9 shows the general circuit structure for the shifter. In this figure, there are only four shift distances: 1 bit, 8 bits, 16 bits and 32 bits. In the VHDL model, the number of shift distances, four or eight, as well as the distances are set through generic parameters before synthesis. This allows the designer to trade shifting capabilities for speed and area.

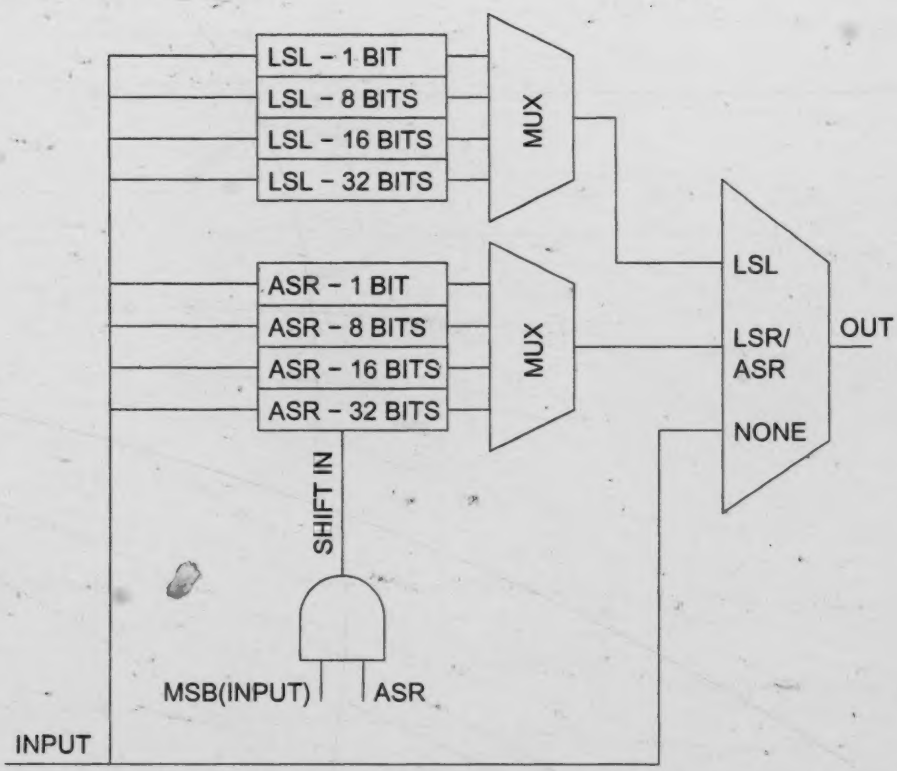



Figure 3.9: Shifter circuit

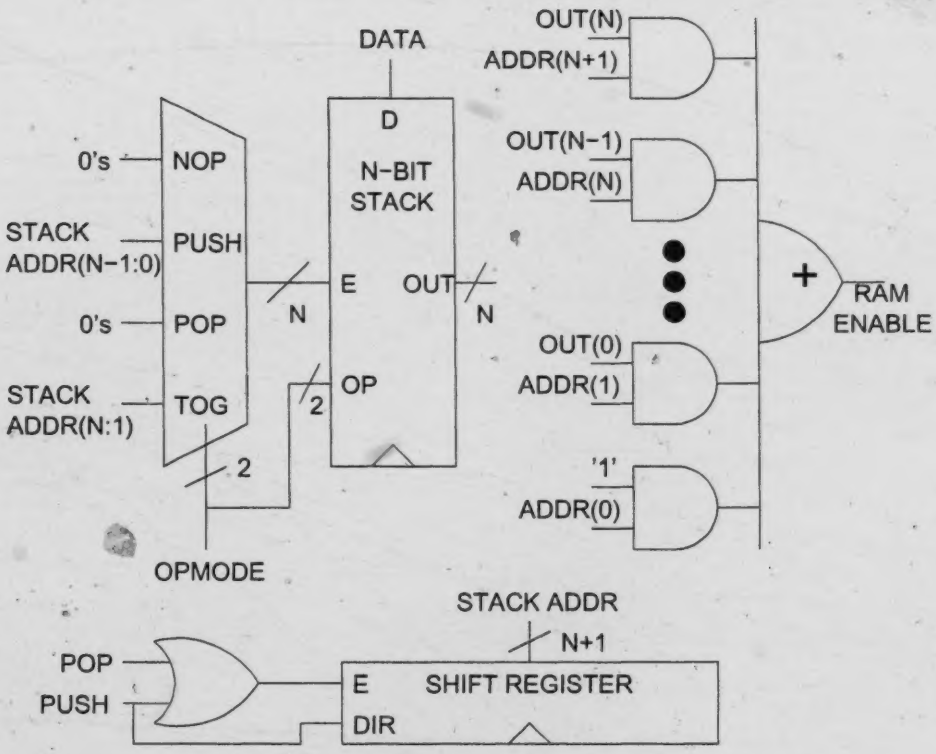



Figure 3.10: Stack circuit

### 3.3.2 Stack

The stack circuit is shown in Figure 3.10. The operating mode of the stack is defined in Table 3.1. This circuit maintains an (N+1)-bit stack address that is reset to

Table 3.1: Stack Operation Codes

| Operation | OpMode |
|---|---|
| NOP | 00 |
| PUSH | 01 |
| POP | 10 |
| TOGGLE | 11 |

"0...01". A bidirectional shift register is used in place of a counter to increment and decrement the stack address. In this way, the stack address can be directly used as enables to each of the stack elements, as well as a mask for the outputs of each stack element. The stack address will only be shifted during a PUSH or POP operation. A

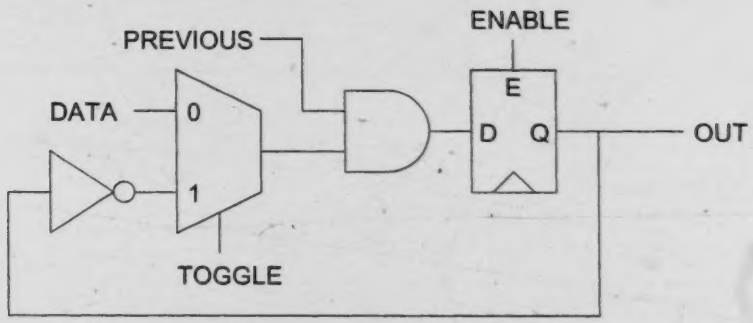



Figure 3.11: Stack element circuit

PUSH operation will shift the address left, while a POP will shift it right. A TOGGLE operation does not modify the stack address. If the processor is configured to POP when the stack is empty or PUSH when the stack is full, the stack address is shifted out of the shift register resulting in a stack address of all zeros. Since the stack address is used to enable the individual stack elements, this will result in not being able to change the state of the stack. The processor cannot recover from this situation because shifting a stack address of all zeros either left or right still results in a stack address of all zeros.

A POP and NOP operation do not require updates to the stack elements so all the bits of the enable bus are set low. For the PUSH operation, the lower N bits of the address are used to enable the stack (all but the most significant bit). Since the stack address always points to the first free element, which is used for a PUSH operation, the TOGGLE operation requires separate addressing. By using the stack address offset by one position to the left, the stack pointer is effectively decremented during the TOGGLE operation.

The RAM enable signal is determined by masking the stack address with the stack output ORing all the bits together. The first address bit always corresponds to enabled because it represents the state when the stack is empty.

By analyzing the stack design, particularly noting that a stack element is only enabled during a PUSH or TOGGLE, the design of the stack element may be simplified. Figure 3.11 shows the circuitry required for a stack element. A single 2-1 multiplexer is used to selectively push the data input or toggle the output. The con-

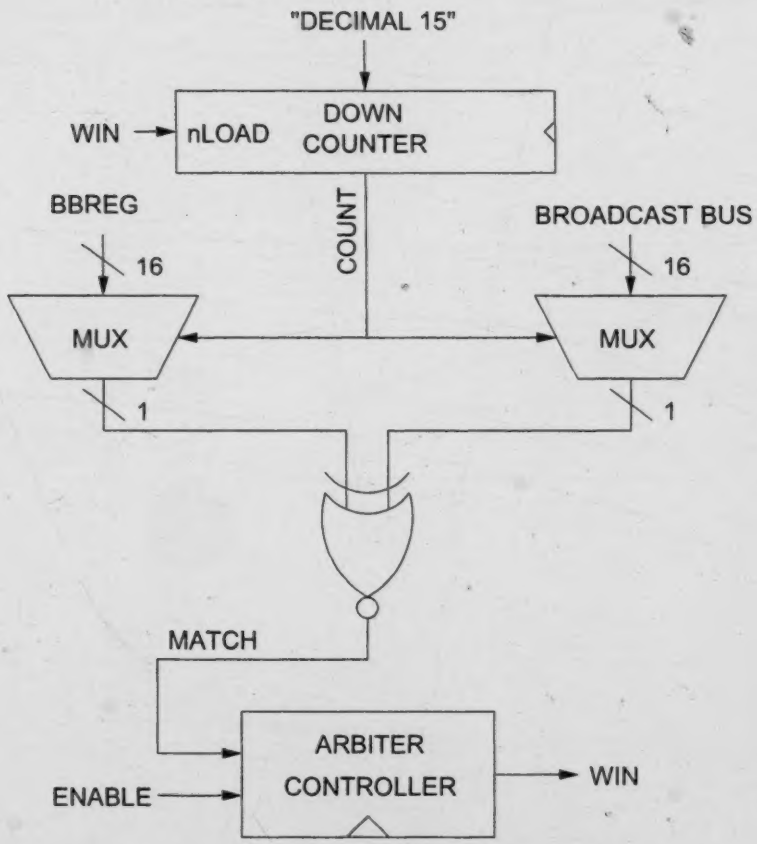



Figure 3.12: Arbiter circuit

trol for the multiplexer can simply be the most significant bit of the operation code shown in Table 3.1 since it is unique between a PUSH and a TOGGLE instruction. As previously pointed out, selecting either input during a NOP or POP instruction is not an issue since the element is disabled. Finally, the new data value to be stored is ANDed with the previous stack element output. This ensures that if the stack was previously disabled, it remains disabled regardless of the data pushed onto the stack, or the data value that is toggled.

### 3.3.3 Arbiter

The arbiter is designed as a multi-cycle unit as shown in Figure 3.12. The first cycle enables the arbiter and causes the WIN signal to go high. The next sixteen

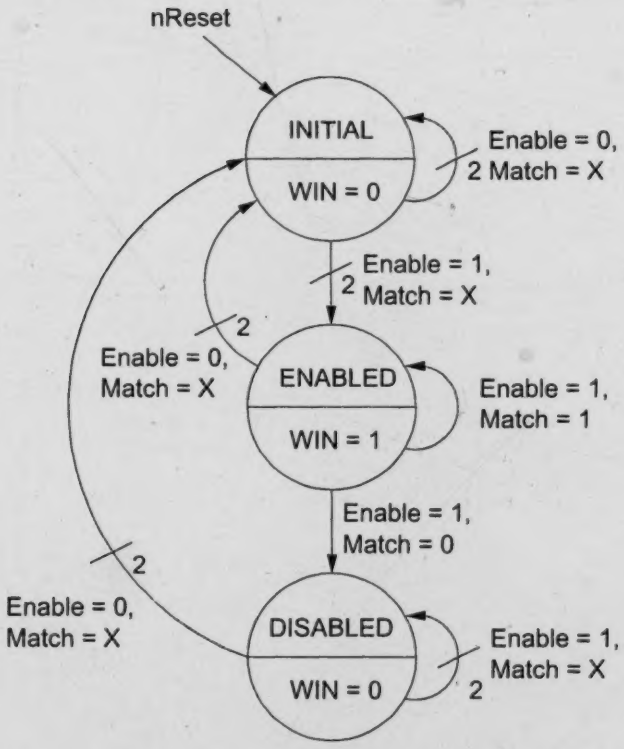



Figure 3.13: Arbiter controller state diagram

cycles compare the bits, indexed by the down counter, of the broadcast bus and the broadcast bus register. A match is detected with the two-input XNOR gate. The counter is initialized with decimal value 15 when the WIN signal is low.

Figure 3.13 is the state diagram for the arbiter controller. Reset causes the arbiter to stay in the initial state and disables the WIN signal. The arbiter will stay in this state until it is enabled and enters the enabled state. The arbiter will stay in the enabled state so long as both the enable signal and the match signal are high. Once the match signal goes low, the arbiter will move to the disabled state and lower the WIN signal. The arbiter will stay in the disabled state regardless of the match signal. If the enable signal goes low, the arbiter returns to the initial state.

Table 3.2: Area and Speed Results for PE Components

| Component | Xilinx VirtexE V2000EFG680-6 | | Xilinx Virtex II Pro 2VP125FF1704-7 | | Standard Cell 0.18 $\mu$m | |
|---|---|---|---|---|---|---|
| | Area (LUTs) | Speed (ns) | Area (LUTs) | Speed (ns) | Area ($\mu m^2$) | Speed (ns) |
| Multiplier | 592 | 9.7 | 32 | 4.0 | 34,090 | 2.6 |
| Adder | 48 | 3.9 | 48 | 3.0 | 7,029 | 4.7 |
| Comparator | 70 | 8.4 | 70 | 4.4 | 1,098 | 2.1 |
| Memory [a] | 0 | 3.5 | 0 | 1.8 | 38,093 | 2.0 |
| Shifter | 548 | 8.1 | 548 | 3.5 | 10,213 | 2.4 |
| Stack | 50 | 3.4 | 50 | 1.3 | 2,720 | 1.0 |
| Arbiter | 37 | 6.4 | 35 | 2.7 | 1,956 | 1.8 |
| Multiplier [b] | 592 | 14.2 | 32 | 4.0 | 21,166 | 3.95 |

[a] FPGA times are sequential delays from datasheets [41] [40], standard cell times are from the TSMC foundry [36].

[b] Combinational multiplier (no pipeline register)

## 3.4 Results

### 3.4.1 Component Results

Table 3.2 shows the area and speed for each of the major components when synthesized for different technologies. The Xilinx VirtexE device is the FPGA used in the Integrator logic module. The Virtex-II Pro is Xilinx's most recent FPGA supported by the FPGA Compiler II tool and represents an FPGA that is enhanced with embedded multiplier cores. The final column represents the results from an ASIC implementation in 0.18$\mu m$ process technology.

The numbers in this table represent the maximum delay for the combinational logic only. Neither the delay through the input and output pads, nor the routing delay to/from the input/output pads is included in the times. In the case of register-based designs, such as the pipelined multiplier, the arbiter and the stack, the maximum combinational logic delay between registers is reported. The set-up times and propagation times for the registers, as well as routing delays to and from the registers are not included.

The first thing to notice in this table is that the multiplier and the shifter are the two largest components. The multiplier is also the slowest component, followed by

the comparator, and the shifter. However, when the design is migrated to the Virtex II Pro FPGA, which contains multiplier cores, the multiplier consumes very little of the reconfigurable logic area and is much faster. In fact, as the final row indicates, a non-pipelined multiplier is the same speed as a pipelined multiplier. This is because the multiplier core for the Virtex Pro is an unpipelined 18-bit multiplier. Therefore, pipelining the multiplier does not make it faster.

The final column displays the results for an ASIC implementation using standard cells in a $0.18\mu m$ process technology. As expected, most of the components are faster than the corresponding FPGA implementations. Again, the multiplier and the shifter are the two largest components in the processor (the memory is not considered part of the PE). Unlike the FPGA implementations, however, the shifter is approximately one-third of the size of the multiplier. The adder is the next largest component, and is also the slowest of the standard cell implementations. By sacrificing area for speed, the adder can be synthesized to a much faster implementation if processor performance is critical. The non-pipelined multiplier shows that it is approximately 62% of the area of a pipelined multiplier and is fast enough to allow for a 200 MHz processing element.

### 3.4.2 Critical Paths

Using the results for the individual modules, three critical paths can be identified in the original PE architecture. These are shown as dashed paths in Figure 3.14. Critical paths are identified by taking the largest progagation delays in Table 3.2 and tracing the input to the module through the most combinational logic to the output of a register. Similarly, the output is traced to the input of a register.

The three longest delays for the two FPGA implementations are the multiplier, the comparator and the shifter. For the ASIC implementation, the longest delays are the adder, the multiplier and the shifter. However, the adder and shifter are part of the same combinational logic path, so will only result in a single path. Therefore, the fourth longest delay in the ASIC implementation, the comparator, is used for the final critical path. This also makes the top three critical paths independent of the implementation.

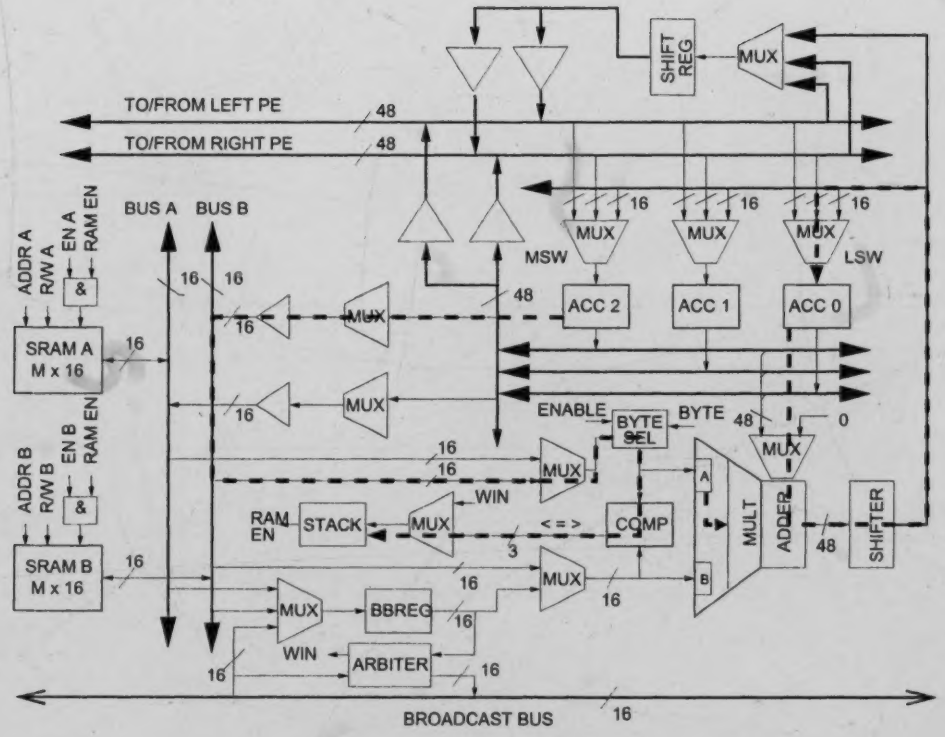



Figure 3.14: Critical paths in the processing element

Table 3.3: PE Critical Paths (For Xilinx VirtexE)

| Path | Delay |
|---|---|
| Acc. output to acc. input through adder/shifter | 22.9 ns |
| Acc. output to stack through comparator | 20.9 ns |
| Input A to mid-multiplier register | 18.3 ns |

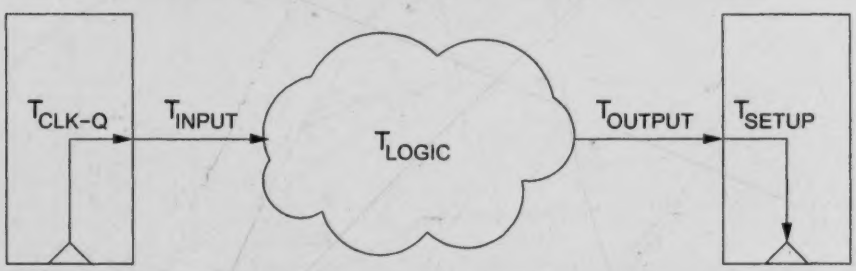



Figure 3.15: Digital circuit delays

The module with the longest delay for the VirtexE FPGA is the multiplier. This is a short path that only goes back to the output of the A register and ends at the input of the middle pipeline register in the multiplier. If instead a non-pipelined multiplier were to be used, the critical path would extend to the product register of the multiplier. The second longest delay is the comparator module in the VirtexE FPGA. This critical path originates from the output of an accumulator register, ACC2 in Figure 3.14, to the input of the stack register. Lastly, the third critical path is through the adder and shifter. This path originates from the output of an accumulator register and stops at the input of an accumulator register (ACC0 in Figure 3.14). The delays through these paths for the Xilinx VirtexE FPGA are displayed in Table 3.3.

Unlike the delays for the modules, these numbers include routing delays and register propagation delays and set-up times. Figure 3.15 shows the different delays inside the logic path. The $T_{CLK-Q}$ delay is the propagation delay through the register. $T_{INPUT}$ is the routing delay from the output of the driving register to the input of the logic. $T_{LOGIC}$ is the delay through the combinational logic. $T_{OUTPUT}$ is the routing delay from the output of the logic to the input of a register. Finally, $T_{SETUP}$ is the time the data must arrive at the register before the rising edge of the

clock. The total delay is then the sum of these delays, shown in Equation 3.3.

$$T_{DELAY} = T_{CLK-Q} + T_{INPUT} + T_{LOGIC} + T_{OUTPUT} + T_{SETUP} \quad (3.3)$$

Before leaving the discussion of critical paths, it is important to discuss the arbiter circuit. While the arbiter circuit is not one of the slowest components in Table 3.2, one of its inputs does not come directly from a register—the wired-AND broadcast bus. What this means is that Equation 3.3 for total delay in a logic path does not hold since the input is not directly from a register. Instead, an additional delay term that describes the delay through the wired-AND broadcast bus must be added in. This delay is proportional to the number of PEs in the DSP-RAM system. Therefore, the arbiter delay must be monitored as the number of PEs increases to ensure that it does not dominate the maximum operating frequency for the PE.

### 3.4.3 PE Results

As mentioned earlier, the VHDL model allows for generic parameterization for specific components in the PE:

1. Four or eight bit position shift distances
2. Depth of stack
3. Include shift register circuitry
4. Include arbiter circuitry
5. Number of overhead bits

The two different shift distances, four or eight, are used to reduce the size and complexity of the shifter circuit. As mentioned previously, the shifter is a critical component in the PE in respect to both the speed and area. Reducing the impact, by only allowing four unique shift values versus eight, provides speed and area savings that may be critical to a design.

The depth of the stack is configurable to allow for different nesting depths in the program. Increasing the depth of the stack increases the area as additional stack

Table 3.4: Area and Speed Results for PE Implementations

| PE Configuration | Xilinx VirtexE V2000EFG680-6 | | Xilinx Virtex II Pro 2VP125FF1704-7 | | Standard Cell 0.18 $\mu$m | |
|---|---|---|---|---|---|---|
| | Area (LUTs) | Speed (ns) | Area (LUTs) | Speed (ns) | Area ($\mu m^2$) | Speed (ns) |
| Original | 1617 | 22.9 | 1320 | 10.5 | 90,342 | 9.63 |
| 4 shift distances | 1403 | 20.9 | 1106 | 9.8 | 84,935 | 7.93 |
| 2 deep stack | 1583 | 22.9 | 1286 | 10.5 | 88,431 | 9.63 |
| No shift register | 1471 | 22.8 | 1174 | 10.1 | 80,052 | 9.45 |
| No arbiter | 1563 | 22.9 | 1266 | 10.5 | 88,081 | 9.63 |
| 1 overhead bit | 1423 | 20.9 | 1126 | 9.6 | 82,231 | 9.19 |
| Minimum-sized | 1033 | 19.7 | 736 | 8.9 | 65,225 | 8.26 |

elements are required, but has little or no impact on the speed of the processor since it is mostly a sequential module.

The shift register circuitry can be removed before synthesis. If the designs that will be run on the PE do not use the shift register then the shift register, and all the circuitry it requires may be removed. Similarly, the arbiter circuitry can be selectively removed to save area as well. For the image processing functions discussed in this thesis, both the shift register and the arbiter are not necessary. However, they are used in other algorithms so the ability to optionally synthesize a DSP-RAM to support them is required.

Finally, the number of overhead bits that the accumulator maintains is configurable from 1 to 16 bits. This reduces the size of the adder and the shifter and helps increase the speed of the processor by reducing the critical path through these two components. When reducing the number of overhead bits, the size of the accumulator remains fixed at 48 bits. This is because ACC2 is used in some algorithms as an independent 16-bit storage location.

Table 3.4 shows the area and speed results for the entire PE with different generic configurations. The first row is the model for the original PE shown in Figure 3.3. The last row presents the results for a PE that sets all parameters to achieve a minimum-sized PE. The rest of the rows assume that only the indicated modification was made to the PE.

For a VirtexE FPGA, synthesizing a minimum-sized PE results in an area savings of 36%. The speed of the original FPGA is dictated by the adder/shifter critical path. The delay through the shifter is lowered by reducing the number of shift distances. The delay through both the shifter and the adder is reduced by using fewer overhead bits. When these modifications are performed, the critical path switches to the path through the comparator logic. Removing the shift register circuitry has only a slight impact on the delay, due to a reduced load seen on the output of the shifter.

# Chapter 4

# Image Processing Experiments on DSP-RAM

Of the image processing kernels discussed in Section 2.3, three will be considered in more detail: thresholding, Sobel dx, and dilate. The thresholding algorithm is the simplest kernel. It does not require any inter-PE communication and requires only a single comparison instruction. The Sobel dx operator requires inter-PE communication to collect the neighbourhood pixels. The algorithm then performs the dot product between the nine pixel elements and the Sobel mask. Finally, the dilate kernel also requires inter-PE communication, but uses the comparator and hardware stack to determine the maximum pixel value. DSP-RAM source code for all the kernels is provided in Appendix D.

## 4.1 Thresholding

The threshold operation uses the comparator and stack to determine if a pixel value stored in a memory bank is greater than a given threshold. If the pixel is greater than the threshold value, it will be replaced with a constant maximum value; otherwise, it will be replaced with a constant minimum value. Since the replacement values are independent of the pixel value, they can be stored in the accumulator registers outside of the main processing loop. This saves cycles in the algorithm by avoiding the high-latency ALU pipeline during processing.

The maximum and minimum values are stored in the ACC1 and ACC0 registers. This is accomplished by using the broadcast bus pipeline shown in Figure 3.4. Note

that the maximum and minimum values do not have to be stored back to memory, so the final memory cycle in the original memory pipeline is removed. In order to store to ACC1 instead of ACC0, the shifter is used to perform a logical shift left a distance of 16 bits. The minimum value stored in ACC0 is not lost because ACC0 is not enabled when the maximum value is stored. The constant threshold value is also stored outside of the main processing loop. This is done by broadcasting the threshold value to all PEs and storing it into the broadcast bus register. This can be done at the same time that the minimum and maximum values are traversing through the multiply-accumulate pipeline. Figure 4.1(a) depicts the state of relevant portions of the processing element after the initial set-up has been completed.

Using Equation 3.1, modified to remove the memory cycle, we can compute the number of cycles required to set-up the PE for thresholding. Since only two values, the minimum and maximum, have to traverse through the pipeline, the $N_{CONSTANTS} = 2$. Therefore, the number of cycles required to set-up for a threshold operation is six.

Once the processor has been set-up, each pixel stored in the local PE memory must be tested against the threshold value stored in the broadcast bus register. Since each 16-bit word of local memory contains two distinct 8-bit pixel values, the byte-select hardware is required to select one of the pixels. At the same time, the result of the greater-than comparison is pushed onto the stack. Figure 4.1(b) shows the hardware path for the first cycle of the threshold operation.

On the next cycle, if the PE is enabled, the maximum value will be stored into the new image location, as depicted by Figure 4.1(c). If the PE is not enabled, nothing will be written back to memory. On this same cycle, the stack is toggled—if the PE was previously disabled, it will be enabled on the next clock cycle.

Cycle three, shown in Figure 4.1(d), will store the minimum value provided that the PE is enabled. Also, the contents of the stack are popped since the threshold operation is complete for that pixel. These three cycles are then repeated for the next pixel by reconfiguring the byte select hardware. This gives a total of six cycles required in the main computation loop.

The first step in determining how many cycles are required for the algorithm

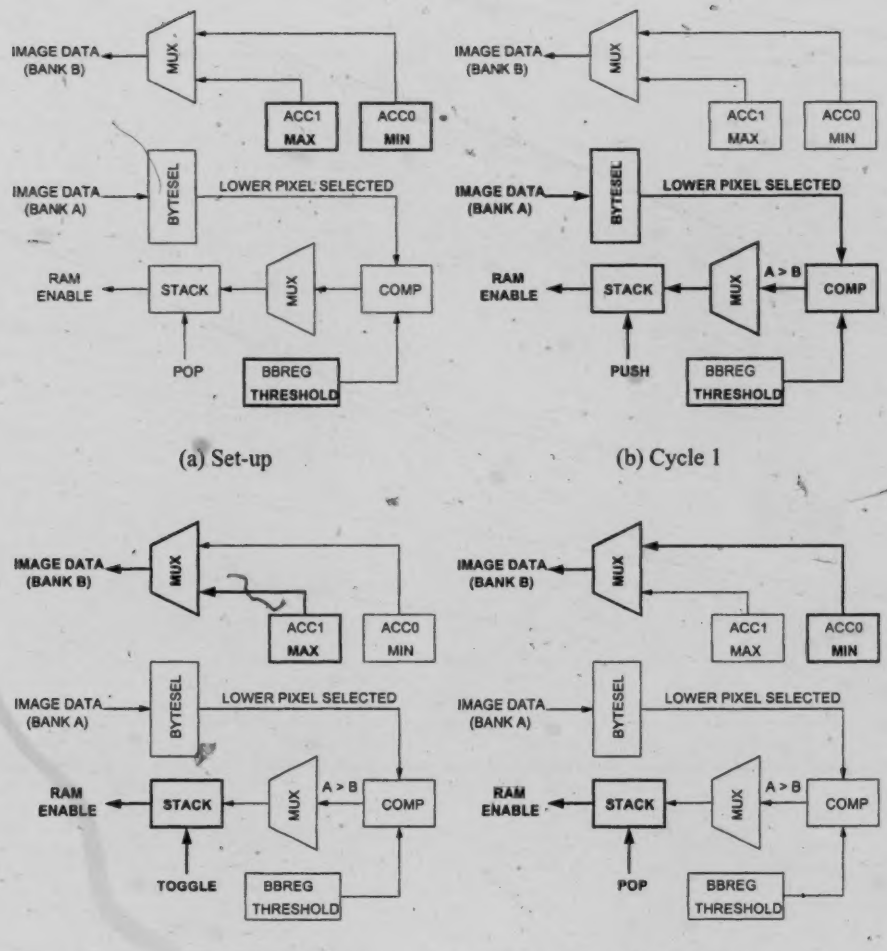



Figure 4.1: Threshold operation on DSP-RAM

is to determine how many 16-bit pixel-pairs are stored on each processor. Given the width (W) and height (H) of an image we define in Equation 4.1 the number of pixel-pairs, $N_{PIXELS}$ stored in each PE.

$$N_{PIXELS} = \frac{W \times H}{2 \times N_{PE}} \tag{4.1}$$

Using this result, Equation 4.3 can be used to compute how many cycles are required given the image size and the number of PEs.

$$N_{CYCLES} = 6 + 6 \times N_{PIXELS} \tag{4.2}$$

$$N_{CYCLES} = 6 + 3 \times \frac{W \times H}{N_{PE}} \tag{4.3}$$

When the image width is twice the number of PEs, this reduces to the value shown in Equation 4.4.

$$N_{CYCLES} = 6 + 6 \times H \tag{4.4}$$

## 4.2 Inter-PE Communication

The remaining image processing kernels involve inter-PE communication. Due to the complexity of transferring neighbourhood pixels between PEs, the networking portion of the algorithm is treated separately. For some algorithms, such as dilate and erode, we will show that this separation results in sub-optimal performance.

Data that is to be transferred to a neighbouring PE must reside in either the accumulator register or the shift register. Both of these 48-bit registers can hold three 16-bit data values. The image processing algorithms use the accumulator register since each of the 16-bit accumulator registers has a separate enable signal, whereas the shift register is treated as a single 48-bit register.

Data is transferred from memory into the accumulator registers by multiplying the data value by the stored constant one. To place a value in ACC2 or ACC1 instead of ACC0, the result is shifted left 32 or 16 bits respectively. Table 4.1 shows how pixel data flows through the multiplier pipeline. As shown in the table, this procedure takes a total of six clock cycles.

Once the data is stored in the accumulator, it can be shifted over the 48-bit shift bus. The incoming data from the neighbouring PE overwrites the three pixel

Table 4.1: Inter-PE Communication Pipeline

| Clock cycle | M0 | M1 | M2 | ACC2 | ACC1 | ACC0 |
|---|---|---|---|---|---|---|
| 1 | Pixel 0 | | | 0 | 0 | 0 |
| 2 | Pixel 1 | Pixel 0 | | 0 | 0 | 0 |
| 3 | Pixel 2 | Pixel 1 | Pixel 0 | 0 | 0 | 0 |
| 4 | | Pixel 2 | Pixel 1 | 0 | 0 | Pixel 0 |
| 5 | | | Pixel 2 | 0 | Pixel 1 | Pixel 0 |
| 6 | | | | Pixel 2 | Pixel 1 | Pixel 0 |

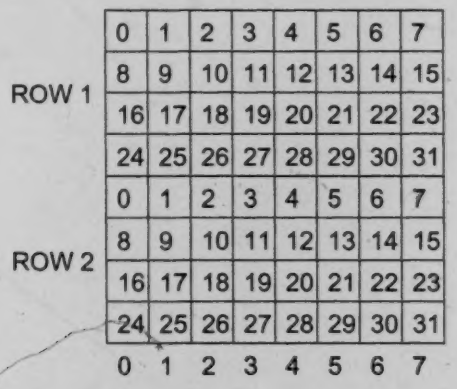



Figure 4.2: Image width spans across four memory rows

values stored in the accumulator. This circuitry was shown in Figure 3.5. The three neighbour pixels are then written to local memory in successive clock cycles. This takes a total of four clock cycles: one for shifting the data, and three for storing the data into local memory.

Next, the original three pixels must again be loaded into the accumulator because the first shift operation over-wrote the contents. The shift process must also be repeated again, but in the opposite direction. In total, it takes 20 clock cycles to collect the nine pixel values.

This result, however, is only for the best case—when an entire row of the image fits in a single memory row of DSP-RAM. According to Equation 4.1, this occurs if the number of PEs in the system is equal to half the width of the image. If it is not, than a single row of image data spans multiple memory rows, as shown in Figure 4.2. In this figure, a single image row spans across four memory rows in the DSP-RAM. This causes a problem for the inter-PE communication routine. Pixel

7, stored in PE(7) in this example, requires pixel 8 for a right neighbour, but it does not have a right-neighbouring PE. Instead, pixel 8 is stored down one memory row in PE(0). Similarly, pixel 8 requires pixel number 7, which is stored up one memory row in PE(7). PE(0) and PE(7) are *edge PEs*. Complicating the problem further is the fact that pixel 0 and pixel 31 do not require this extra effort since these are the natural starting and ending pixels in the image row.

In order to take care of this case, we define the terms *index* to equal the current row, and *offset* to equal the number of rows the image spans. The offset computation is given by Equation 4.5.

$$Offset = \frac{W}{2 \times N_{PE}} \tag{4.5}$$

To start, we examine the last PE in the system. This PE requires data to be transferred from PE(0) for every row except the row that contains the natural ending pixel (pixel 31). This only occurs when the index is a multiple of the offset, which happens only once when processing a single image row. Similarly, PE(0) requires data to be transferred from the last PE in the system for every row except the row that contains the natural starting pixel (pixel 0). This is only the case for the first memory row when processing an single image row.

Data transfer from one edge PE to the other edge PE follows the same algorithm. Pixels are moved from memory into the accumulator registers for all PEs in the same manner as before. Once the data is in the accumulator, it is shifted left when the data transfer is from PE(0) to PE(N). It is shifted right when the data transfer is from PE(N) to PE(0). For both cases, the controller stores the data that is shifted out of the SIMD array, as illustrated by Figure 4.3.

Next, PE(0) or PE(N) is selectively enabled by comparing the PE identification number stored in each processor with 0 or N depending on the algorithm. Once only one of the edge PEs is enabled, the controller injects the stored shift value into the opposite end of the array. The selectively-enabled PE then stores these values into local memory. When the last value is stored, the PE pops the contents of the stack and normal SIMD processing is resumed.

The process of transferring three pixel values from one edge PE to the opposite edge PE takes a total of 12 cycles to complete. Six cycles are required to store the

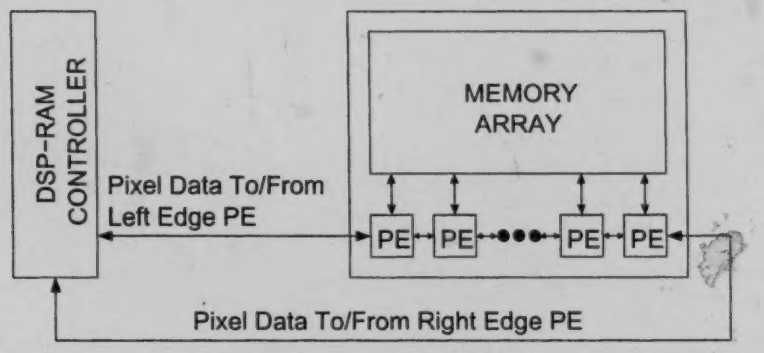



Figure 4.3: DSP-RAM controller stores and injects pixels from edge PEs

three pixel values into the accumulator. One cycle is required to shift the data out of the SIMD array and into the controller (shown in Figure 4.3). The five remaining cycles are for conditionally storing the pixel values back to memory: one cycle for enabling the PE, one cycle for shifting the value in from the controller, and three cycles for writing back to memory.

Equation 4.8 shows how many cycles are required to perform inter-PE communication.

$$N_{CYCLES} = 20 \times Offset + 2 \times ((Offset - 1) \times 12) \quad (4.6)$$

$$N_{CYCLES} = 44 \times Offset - 24 \quad (4.7)$$

$$N_{CYCLES} = 22 \times \frac{W}{N_{PE}} - 24 \quad (4.8)$$

The constant 20 is the number of cycles required to collect the neighbours for the interior PEs. This has to be repeated for each memory row over which a single image row spans. The second part of the equation is multiplied by two to account for transferring data from PE(N) to PE(0) and from PE(0) to PE(N). This 12-cycle process is not required for each memory row that comprises a single image row. The first memory row does not need to transfer data from PE(0) to PE(N) because it contains the left edge pixel. Likewise, the last memory row does not need to transfer data from PE(N) to PE(0) since it contains the right edge pixel. Therefore, the computation is required only for offset - 1 memory rows. When the offset equals one, the equation simplifies down to a fixed 20 cycles.

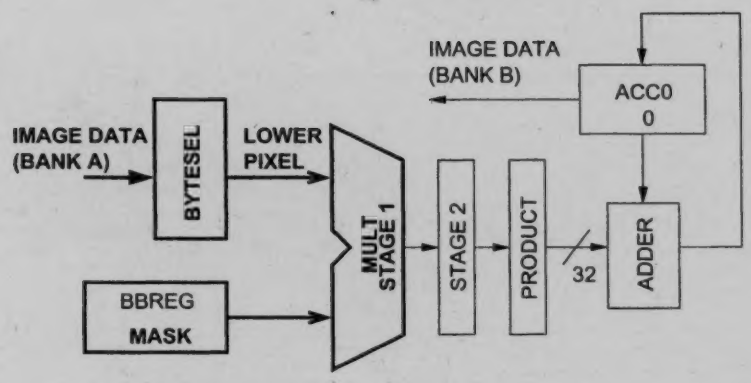
## 4.3 Sobel Dx Operator

The Sobel kernel performs a vector dot product between the neighbourhood pixels and a mask array that is constant across all PEs. Once the neighbourhood pixels are collected, they are multiplied by the constants in the mask array and the product is accumulated in the accumulator registers. Figure 4.4(a) shows the portions of the processor used for the Sobel operators for the first multiplication. At this point, the original pixel value stored in bank A and the mask value stored in the broadcast bus register will enter the first stage of the multiplier pipeline. Figure 4.4(b) shows the processor with a full pipeline during execution of the Sobel kernel. Here, a new pixel value and mask value enter the start of the multiplier pipeline while the current sum of all the previous products is being stored in the accumulator at the end of the pipeline. All the intermediate pipeline stages are full.

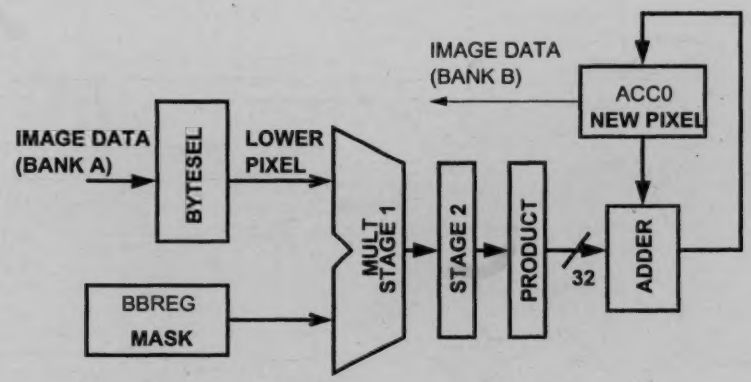
The final configuration for the PE is shown in Figure 4.4(c), where the 16-bit signed result is stored back to memory. At this point, the new 16-bit pixel value can be stored back into bank B. This multiply-accumulate process is then repeated for the second pixel value stored in the memory bank. Given that the Sobel operator is strictly using the broadcast bus to memory pipeline shown in Figure 3.4, we would expect the number of cycles to follow Equation 3.1. The number of constants would be nine, resulting in 14 cycles. Since the process has to be repeated, you would expect a total of 28 cycles. However, for the second iteration, the five overhead cycles required to fill the pipeline can operate in parallel with the last five cycles of the first iteration. So Figure 4.4(c) actually represents the state of the processor at the end of processing the second pixel, not the first. By coding the algorithm in this way, the number of constants in Equation 3.1 is actually 18 instead of 9, and the total number of cycles is 23 instead of 28. By combining Equations 3.1 and 4.8, the number of cycles required to execute the Sobel kernel on one pixel-pair is given by Equation 4.10.

$$N_{CYCLES} = (22 \times \frac{W}{N_{PE}} - 24) + (5 + 18) \tag{4.9}$$

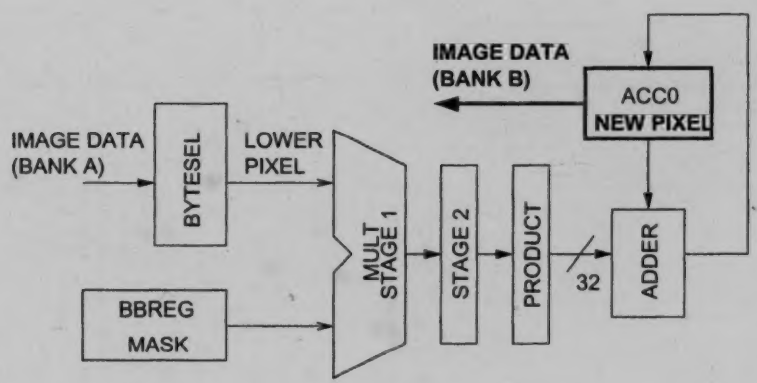
$$N_{CYCLES} = 22 \times \frac{W}{N_{PE}} - 1 \tag{4.10}$$




(a) Set-up




(b) Cycle N




(c) Memory

Figure 4.4: The Sobel operation on DSP-RAM

To find the total number of cycles required to process an image, Equation 4.10 must be multiplied by the number of pixel-pairs stored in a PE (Equation 4.1). The result is expressed in Equation 4.11.

$$N_{CYCLES} = \frac{W \times H}{2 \times N_{PE}} \times (\frac{22 \times W}{N_{PE}} - 1) \tag{4.11}$$

Assuming that the width of the image is twice the number of processors, Equation 4.11 reduces to Equation 4.12.

$$N_{CYCLES} = H \times 43 \tag{4.12}$$

This equation shows the benefits of parallel execution. Provided that the number of processors increases with the width of the image, the number of cycles required to peform the computation depends only on the height of the image, not the width.

## 4.4 Dilate

The dilate kernel computes the maximum pixel value from among the nine neighbourhood pixels. Once all the neighbourhood pixels have been collected, the algorithm starts by moving the first pixel value into the maximum memory location. This uses the ALU pipeline in Figure 3.7 and therefore requires five cycles. Next, each of the remaining eight pixel values is compared against the maximum pixel value. If a pixel value is greater than the current maximum, the greater pixel value is loaded into the maximum memory location. The comparison operation takes a total of six clock cycles. The first cycle performs the comparison operation and selectively enables the PE. The remaining five cycles come from moving the data through the ALU pipeline. Since eight pixel values must be compared against the maximum, 48 clock cycles are required. Once the maximum value is found, it must be saved into the new image. This requires sending the maximum pixel value through the ALU pipeline to be stored in the new image memory location resulting in another five cycles. At this point, the algorithm must be repeated for the second pixel. The initial loading of the first pixel value into the maximum memory location can be overlapped in the ALU pipeline with storing the result pixel. Therefore, this loading operation only requires a single cycle compared to five cycles for the first

pixel. The remaining portions of the algorithm require the same number of cycles. Using these results and Equations 4.8 and 4.1, the total number of cycles required for the dilate kernel can be expressed by Equation 4.13.

$$N_{CYCLES} = \frac{W \times H}{N_{PE}} \times (11 \times \frac{W}{N_{PE}} + 44) \tag{4.13}$$

When the width of the image is twice the number of PEs, Equation 4.13 reduces down to Equation 4.14:

$$N_{CYCLES} = H \times 132 \tag{4.14}$$

which, like the Sobel operator, is independent of the width of the image.

The results for the dilate kernel are almost three times worse than for the Sobel kernel. The relatively poor performance comes from two sources: the communication algorithm and the ALU pipeline. The communication algorithm used is the same for all the kernels. This code re-use causes a sub-optimal design in the dilate operation. First, each PE performs eight comparisons to find the maximum pixel value from the nine neighbourhood pixels. However, if each PE first found the maximum value from its three local pixels, it would require only two comparisons. The resulting single pixel could then be shifted to both the left and right neighbouring PEs. At this point, each PE has to again find the maximum of three pixel values, which requires two more comparisons. This way, the total number of comparisons is four—half the number originally required when using the common neighbourhood collection.

The second source of the high cycle count is caused by the ALU pipeline. Whenever data must be moved from one memory location to another, the only route is through the ALU pipeline. Listing 4.1 shows pseudo-code for comparing two pixel values in DSP-RAM. The CMP.GT instruction compares the pixel 'p1', stored in memory bank A, with the current maximum stored in memory bank B. If 'p1' is greater than the maximum value, the conditional flag GT is set. Both operands to this instruction are source operands—the CMP.GT instruction only reads memory locations, it does not write to them. The MOV instruction conditional transfers the data from the source operand 'p1' to the destination 'max'. This 5-cycle operation causes a Read-After-Write (RAW) hazard. The third instruction cannot execute

Listing 4.1: DSP-RAM data hazard

```
      CMP.GT   bankA(p1),bankB(max) ; 1 cycle
[GT]  MOV      bankB(max),bankA(p1) ; 5 cycles
      NOP
      NOP
      NOP
      NOP
      CMP.GT   bankA(p2),bankB(max) ; 1 cycle
[GT]  MOV      bankB(max),bankA(p2) ; 5 cycles
```

until the MOV instruction has completed. This means NOP instructions must be inserted into the pipeline to ensure that the MOV instruction completes before the next comparison. It is important to realize that the high latency of the MOV instruction is caused by the pipelined MAC unit, even though functionally, the MOV instruction performs no multiply or accumulation function!

## 4.5 Results

The performance of the DSP-RAM architecture was compared against the performance of a 1.6 GHz Pentium 4 desktop computer with 512 MB of RAM. The implementations for the image kernel used the Integrated Performance Primitives (IPP) library from Intel. This library makes expert use of the SSE2 instruction set available to the Pentium 4 architecture. Each of the algorithms was averaged over 10,000 runs in an attempt to capture average loading characteristics on the Linux machine. During the simulation runs, only one user was logged in via the console, not using the X display. Before continuing, it is worth pointing out that we are not trying to replace the Pentium 4 with DSP-RAM. DSP-RAM is intended to be used as a co-processor in conjuction with a sequential host processor. Large memory-intensive parallel applications may be farmed out to a low-power DSP-RAM co-processor to receive similar results as a convential desktop machine, while the host processor is freed up to perform other tasks.

Table 4.2 presents the results for a 512×512 image. These results strictly reflect the simulation time for the core algorithms. They do not include any overhead in-

Table 4.2: DSP-RAM Image Processing Results (512×512 Image)

| Kernel | Pentium 4 (1.6 GHz) | DSP-RAM (50 MHz) | Speed-up |
|---|---|---|---|
| Threshold | 0.4 ms | 0.06 ms | 6.67 |
| Sobel dx | 0.9 ms | 0.44 ms | 2.05 |
| Sobel dy | 0.9 ms | 0.44 ms | 2.05 |
| Dilate | 0.6 ms | 1.35 ms | 0.44 |
| Erode | 0.6 ms | 1.35 ms | 0.44 |
| Box Filter | 1.7 ms | 0.49 ms | 3.47 |
| Median Filter | 0.9 ms | 7.10 ms | 0.13 |

volved in loading data into memory for neither the Pentium 4 nor the DSP-RAM. The DSP-RAM times are derived by multiplying the cycle count by a hypothetical clock frequency of 50 MHz, which represents a cycle time of 20 ns. From Table 3.4, this processor frequency is achievable for a minimum-sized processing element for all technologies presented. In fact, a Virtex II Pro and standard cell implementation could operate at 100 MHz. The standard cell implementation has even been synthesized for 200 MHz operation at the expense of area.

From this table, the threshold operation receives the largest benefit from parallelism. This is due to the lack of inter-processor communication and the simplicity of the algorithm. Next, the box filter and Sobel operators produce speed-ups of 2.05 and 3.47 times, respectively. Both of these algorithms map efficiently onto the DSP-RAM architecture, and few pipeline stalls are introduced. The larger box filter speed-up is caused by the performance difference between it and the Sobel dx kernel on the Pentium 4. The Sobel dx filter coefficients are 0, $\pm 1$, and $\pm 2$. Rather than performing full multiplication when multiplying by 2, the Pentium 4 can use a left shift by one to achieve the same results. This results in a fast Sobel operation whereas the box filter must take the average of the nine pixel values, so an integer division is required at the final stage, increasing the time for this algorithm.

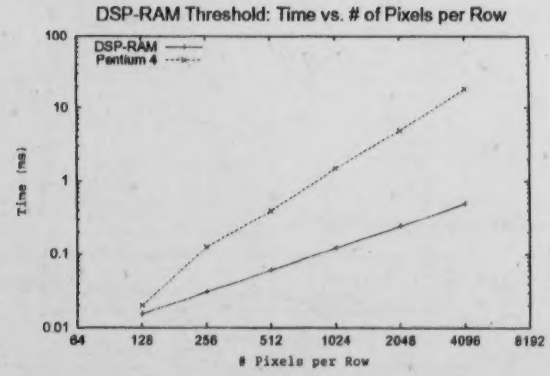
The remaining algorithms, dilate, erode, and median, all have worse performance than the sequential Pentium 4. Dilate and erode suffer from having to move data through the high-latency ALU pipeline. If there were to be a method to by-pass the ALU, the performance of these two operations would increase. The speed-up for dilate and erode is also lower because they are very efficiently implemented on

the Pentium 4. Besides thresholding, these are the two fastest algorithms on the Pentium 4.

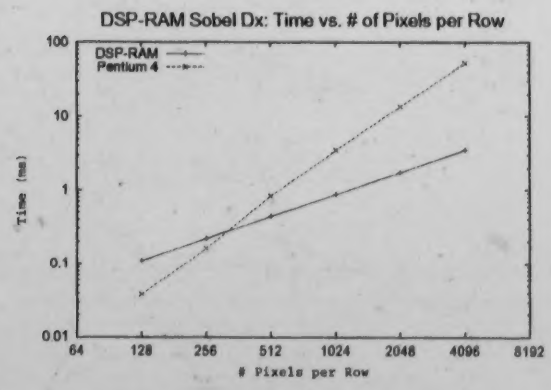
The median filter has the worst performance on DSP-RAM. The median filter implements a sorting algorithm that utilizes the same move command as the dilate operator and therefore experiences the same pipeline problems. Compounding the problem is the number of comparisons required. If you remember, the dilate operation that was implemented required eight comparisons for each pixel, resulting in a total of 16 comparisons. The median filter, on the other hand, requires a total of 80 comparisons! In fact, even if two pixels could be swapped in a single cycle, the median kernel would still only see a speed-up of 0.66.

Figures 4.5(a)-4.5(c) show the results for each of the algorithms as the size of the image increases. The threshold operation on DSP-RAM out-performs the Pentium 4 for all the image sizes used, as shown in Figure 4.5(a). The Sobel filter matches the performance of the Pentium 4 at around 350 pixels per row, while the dilate kernel starts out-performing the Pentium 4 only at around 1300 pixels per row. For all three algorithms, the performance gap between DSP-RAM and the Pentium 4 increases exponentially as the image size increases. This is a typical result for SIMD computing.

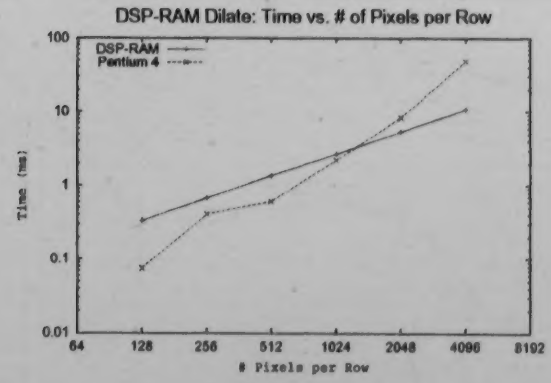
While image sizes of 2048 and 4096 are shown on the graph, these currently are not practical image sizes for DSP-RAM. They are included to help illustrate how SIMD computing benefits from increased parallelism and to show that DSP-RAM will continue to be useful as technology moves forward. However, as long as process fabrication technology follows Moore's law, system capacity and memory resources are going to continue to increase. Further, once(if) the HDTV standard becomes mainstream it will most likely be in existance for a long time because it is a significant and expensive upgrade from current technology. Therefore in one or two years an ASIC DSP-RAM implementation may be able to handle the 1920 × 1080 HDTV resolution where an FPGA might not. However, five to six years from now, and FPGA implementation of DSP-RAM may also be able to process the video stream.




(a) Threshold




(b) Sobel dx




(c) Dilate

Figure 4.5: Computation time versus image size

# Chapter 5

# Architecture Modification Experiments

Chapter 3 described the original DSP-RAM architecture, and Chapter 4 explained how the image processing algorithms are mapped onto it. In this chapter, we investigate two possible modifications to the architecture. The first modification aims primarily to reduce the area of the PE, while minimizing the impact on algorithm performance. The second modification seeks to increase algorithm performance, while having a minimal impact on the size of the processor.

## 5.1 Inter-PE Communication

The first modification made to the PE was to the shift bus. Figure 5.1 shows the path that data must take to be transferred from the local memory of one PE to that of another PE. When three 16-bit values are packed into the 48-bit accumulator register, they can all be transferred in a single cycle to the neighbouring PE. However, the interface to memory is still 16 bits, so it takes three cycles to store the individual values. While it is true, that the total memory bandwidth is 32 bits (both memory bank A and memory bank B could be loaded in parallel), we found that it was much simpler to keep all the data together in a single memory bank. Keeping the data together in a single memory bank the side-steps the structural hazard by using memory bank A as the source and memory bank B as the destination. It also removes the need for the programmer to keep track of which memory bank is being used at each step of the algorithm. Shifting the accumulator and storing the three

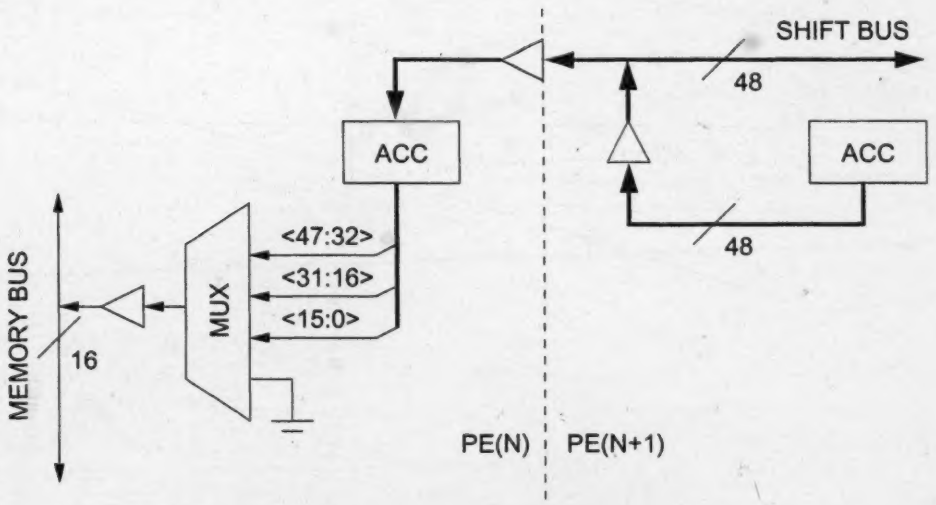



Figure 5.1: Memory transfer between PEs

data values into the PE local memory requires a total of four cycles: one cycle to shift the 48-bit data, and three cycles to store them individually back into memory. In this case, having a wide shift bus provides no advantage because the bottleneck in the system is the memory bus.

By modifying the system to only have a 16-bit shift bus (shown in Figure 5.2), the memory transfer can be efficiently pipelined. Since the width of the shift bus now matches that of memory, a new 16-bit value can be shifted at the same time that the previous value is being stored to memory. This new method of transferring neighbouring data also requires four clock cycles. This allows us to eliminate two-thirds of the interconnect between processors without suffering any performance penalties while transferring data between local memories!

Simply reducing the number of wires between PEs is not all that is required, however. In Figure 3.3, three 3-input multiplexors were required to select whether to load the accumulator with data from the left shift bus, the right shift bus, or the shifter. When the shift bus is reduced to 16 bits, the same data is routed to each of the accumulator registers. Therefore, the selection between the left and right shift bus can be determined using a single 2-1 mux. The remaining choices are to load the accumulator with data from the shifter or data from the predetermined selection

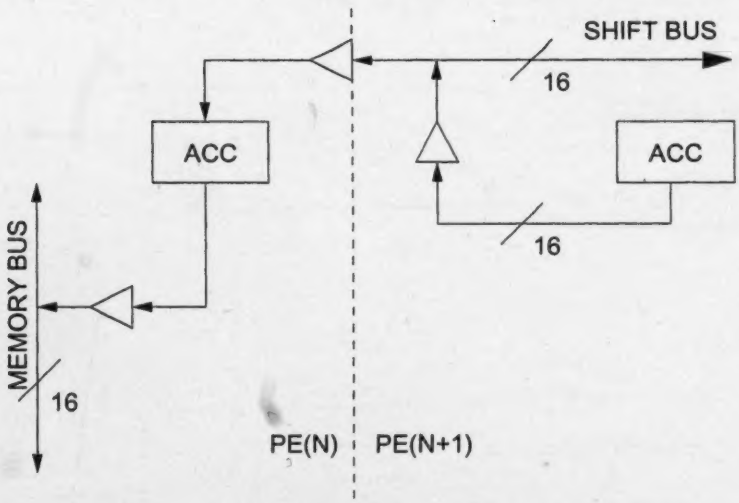



Figure 5.2: Memory transfer between PEs modified to use a 16-bit shift bus

of left or right shift bus (Figure 5.3). This new design requires four 2-1 muxes instead of three 4-1 muxes. Assuming a single 3-1 mux can be implemented using two 2-1 muxes, three 3-1 muxes is equivalent in cost to six 2-1 muxes. Therefore a design that uses only four 2-1 muxes will be smaller and faster.

Just as the number of bits that the shift bus could drive into the accumulator was reduced, the number of bits that the accumulator can drive onto the left and right shift buses is reduced. In the original architecture, a single 48-bit tri-state driver was all that was required to interface the accumulator to the shift bus. Now, a 3-1 mux is needed to select which one of the three 16-bit accumulator registers will be connected to the shift bus. Figure 5.4(a) shows what this design would look like.

While this design would work, it was not what was chosen. Through auditing the hardware utilization for the algorithms, it was noticed that both the RAM A and RAM B tri-state drivers were never on at the same time. This means that since both of the multiplexors are connected identically, and only one of the outputs is used at any given time, one of them is redundant. Figure 5.4(b) shows the final recommended design for this area of the processor.

Like most design decisions, this optimization does not come without drawbacks. Consider the case when the accumulator of Figure 5.1 contains a single 48-bit value

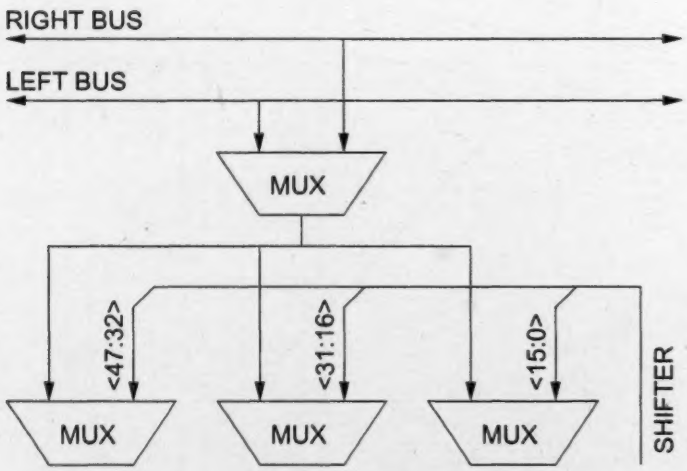
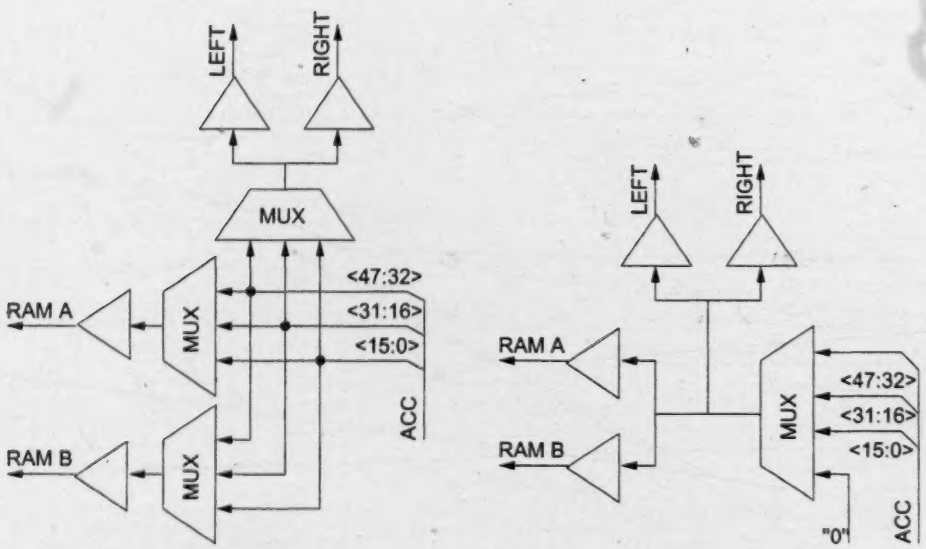



Figure 5.3: Pre-computed left or right shift bus selection




(a) Three 3-input muxes

(b) One 4-input mux

Figure 5.4: Accumulator registers selection

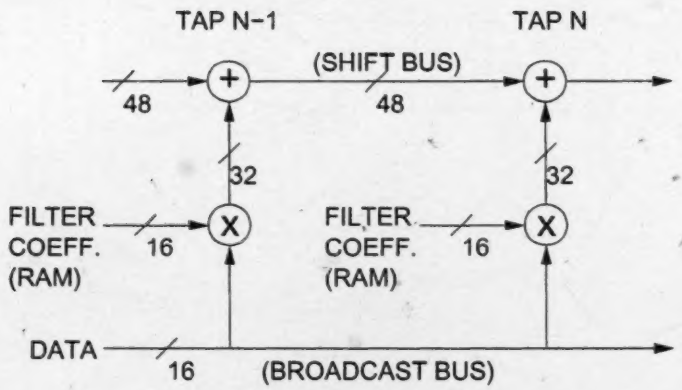



Figure 5.5: FIR filter

that was computed by the PE. This 48-bit value then must be transferred to the neighbouring PE to be used as an input to the adder, not to be stored into memory. This is the case for an FIR filter, shown in Figure 5.5. In this scenario, the 16-bit memory bus does not create a bottleneck so the performance will be impacted for algorithms that use this implementation style.

Another performance hit is seen when streaming data in through the shift bus, as described in Section 3.1.1. In this method, three 16-bit data values can be injected into the DSP-RAM system by the controller every clock cycle. After $N_{PE}$ shifts, each PE in the system contains three data values in its respective accumulator registers. The system then requires three cycles to store all the data values into memory. This resulted in approximately one-third of a clock cycle per data value, as shown in Equation 3.2.

By reducing the shift bus from 48 bits down to 16 bits, the controller can only inject one 16-bit value per clock cycle. This increases the number of cycles required for shifting data to the value specified in Equation 5.1.

$$N_{CYCLES} = 1 + N_{DATA} \tag{5.1}$$

This asymptotically approaches one clock cycle per data value—a performance degradation of almost three times for storing data via the shift bus. While this is a large performance hit, we found that data storage is typically performed outside of the main processing loop, so the loss of performance can be amortorized over the

entire execution time.

## 5.2 Enhanced PE Routing

While reducing the size of the shift bus attacked the size and complexity of the PE, enhancing the routing inside the PE attempts to improve performance for selected algorithms. Of the image processing algorithms studied, the dilate operator and median filter both showed relatively poor performance due to the use of the comparator and hardware stack. Since parallel systems achieve performance benefits through parallel computing, disabling some of the processors decreases the usable computing power. This is a consequence of the restrictive SIMD architecture and can only be avoided by redesigning the algorithm. The second cause, however, is from the design of the DSP-RAM processing element itself, specifically, the ALU pipeline (Figure 3.7). These designs require data to move conditionally through the high-latency ALU pipeline without performing any meaningful computation. This means that the multiplier is simply multiplying by the constant one, with the adder being disabled. Further, the data hazard illustrated in Listing 4.1 forces NOPs to be inserted into the pipeline, which decreases the throughput.

One solution to this problem is to provide an ALU by-pass path from the comparator to memory. In fact, this can be accomplished in the current PE without adding any new components. First, a drive path to the data bus must exist. Looking at the newly constructed data path mux in Figure 5.4(b), the fourth input is unused. Instead of being tied to ground, these inputs can be connected to the 16-bit data value that must by-pass the ALU. Next, the data value must be stored temporarily during the comparison so that it can be written back to memory on the next clock cycle. This can be done by using one of the input registers to the multiplier. The resulting modification is shown in Figure 5.6. One important side-effect in this design is that an ALU by-pass path has been created not only for the memory banks, but also to the shift bus via the common multiplexor design.

Table 5.1 shows the area and speed results for the modified processing element. In this table, the area and speed are compared against the results obtained for a min-

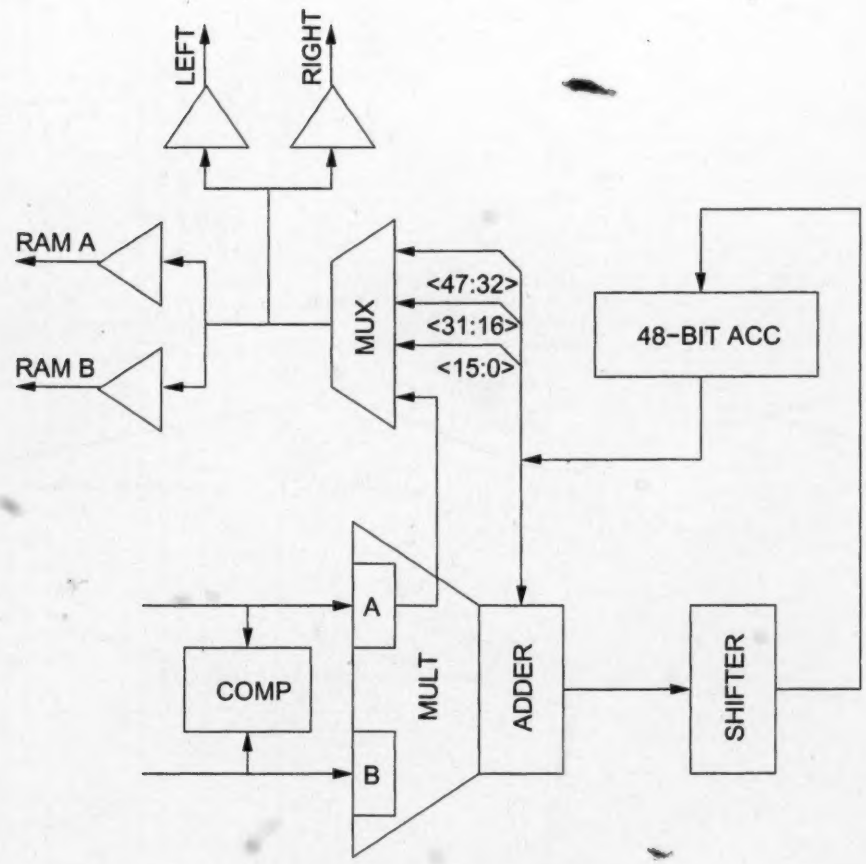



Figure 5.6: ALU by-pass routing

Table 5.1: Area / Speed Results for PE with Enhanced Routing

| **Technology** | **Area / % Decrease** | | **Delay / % Decrease** | |
|---|---|---|---|---|
| Virtex E (V2000EFG680-6) | 924 LUTs | 10.6% | 19.7 ns | 0.0% |
| Virtex II Pro (2VP125FF1704-7) | 627 LUTs | 14.8% | 8.3 ns | 6.7% |
| Standard Cell (0.18 $\mu$m) | 56,431 $\mu m^2$ | 13.5% | 7.82 ns | 5.3% |

imum sized PE synthesized in each of the studied technologies shown in Table 3.4. This shows that we can achieve a 10 to 15 percent decrease in area through these modifications. For the Virtex II Pro and standard cell technology there is also a slight improvement for the longest delay path.

## 5.3 Algorithm Performance

Given the new modifications to the processing element, it is necessary to revisit the design of the image processing algorithms. Obviously the algorithms using the comparison operation will benefit from the changes, but other portions of the design may be able to be modified to benefit even more from the new architectural features.

### 5.3.1 Threshold

The first algorithm was the threshold operation. Since there is no inter-processor communication required for the threshold operation, the modification to the shift bus has no impact on the performance of this algorithm. While this algorithm does use the comparator, the value to be stored back to memory does not depend on the data being compared. That is, the final pixel value is set to a constant maximum value or a constant minimum value, both of which are stored in the accumulator registers outside of the main algorithm loop. Therefore, there are no cycles to be gained by using either modification to PE.

### 5.3.2 Inter-PE Communication

Next, the inter-PE communication section is reviewed. This section will require re-work as it depends primarily on the shift bus. However, since data must travel through the ALU pipeline without being modified, the enhanced routing can also be exploited. Instead of using the MAC unit to store the pixel values into the accumulator, the first cycle stores a 16-bit pixel value into the A-input of the multiplier. The second cycle drives the pixel value left and loads the right neighbour PE's pixel value into ACC0. The third cycle is used to drive the pixel value right and load the left neighbour PE's pixel value into ACC1. It is important to realize that the value

stored in ACC0 cannot be stored back to memory during this clock cycle because of the single multiplexor used in the new design. Therefore, the fourth and fifth cycles are used to store the two values back to memory. This 5-cycle process is repeated for other two pixel values, resulting in a total of 15 clock cycles in the best case. This is 25% improvement over the previous architecture for the inter-PE communication portion of the design.

Like the previous design though, this is only for the best case—when the width of the image is twice the number of processors. When the image is larger the same approach is taken, only this time the enhanced routing can again be exploited. First, the pixel value from PE(0) is extracted from the left side of the DSP-RAM. As was done previously, the shift bus is driven directly from the A-input of the multiplier. The controller will now inject this new pixel value into the right side of the DSP-RAM to be stored in PE(N)'s accumulator. This process is pipelined so that the three pixel values can be transferred from PE(0) to PE(N) in a total of five clock cycles. Next, the last PE is selectively enabled by comparing the PE identification number with N and the three pixel values are stored back to memory in subsequent clock cycles. This process requires a total of 8 clock cycles to complete, and must again be done when transferring the pixel values from PE(N) to PE(0). Compared to the previous architecture, this represents a 33.3% improvement in cycles. Equation 5.4 shows the number of cycles required for inter-PE communication when using the new architectural features.

$$N_{CYCLES} = 15 \times Offset + 2 \times ((Offset - 1) \times 8) \quad (5.2)$$

$$N_{CYCLES} = 31 \times Offset - 16 \quad (5.3)$$

$$N_{CYCLES} = 15.5 \times \frac{W}{N_{PE}} - 16 \quad (5.4)$$

The dominant term has been reduced from 22 in Equation 4.8 to 15.5 in Equation 5.4, an improvement of 29.5%. When the width of the image is twice that of the number of PEs, the number of cycles reduces to 15, an improvement of 25%. This, in fact, is the more common case, where the 29.5% improvement is the theoretical maximum achieved only when the width of the image approaches infinity.

### 5.3.3 Sobel Dx Operator

The sobel Dx operator makes heavy use of the pipelined MAC unit in DSP-RAM. In fact, the core algorithm mapped the most efficiently onto the original architecture of all applications studied. The shift bus and ALU by-pass are not used at all outside of the inter-PE communication module for this algorithm so the number of cycles changes only due to the inter-PE communication, as specified in Equation 5.6.

$$N_{CYCLES} = (15.5 \times \frac{W}{N_{PE}} - 16) + (5 + 18) \quad (5.5)$$

$$N_{CYCLES} = 15.5 \times \frac{W}{N_{PE}} + 7 \quad (5.6)$$

To find the total number of cycles, we again multiply this equation by the number of pixel pairs stored on each PE (Equation 4.1). Assuming that the width of the image is twice the number of processors, Equation 5.6 reduces to Equation 5.8.

$$N_{CYCLES} = \frac{W \times H}{2 \times N_{PE}} \times (\frac{15.5 \times W}{N_{PE}} + 7) \quad (5.7)$$

$$N_{CYCLES} = H \times 38 \quad (5.8)$$

This represents an improvement of 11.6% over the previous architecture, even though the core algorithm did not utilize the modifications.

### 5.3.4 Dilation

The dilate operator is able to directly benefit from the new ALU by-pass circuitry. First, the initial maximum must be loaded with the first pixel value. Using the by-pass path, this takes only two cycles compared to five in the original architecture. Next, each of the remaining eight neighbourhood pixels is compared against the maximum. If a pixel is greater it is stored as the new maximum value. This also only requires two cycles to complete. The first cycle performs the comparison and pushes the result onto the stack, as well as stores the pixel value into the A-input of the multiplier. The second cycle conditionally replaces the maximum value in the memory bank with the A-input, as well as popping the stack. The original architecture required 6 cycles to perform the same function.

Once the maximum has been found, it again requires only two more cycles to move it into the processed image location. This process is repeated for the second

pixel stored in the upper byte of the memory location. Using Equation 5.4 for the cycles required for collecting the neighbourhood pixels, and these new cycle counts, we can derive Equations 5.10 and 5.10.

$$N_{CYCLES} = 15.5 \times \frac{W}{N_{PE}} - 16 + 2 \times (2 + 8 \times 2 + 2) \quad (5.9)$$

$$N_{CYCLES} = 15.5 \times \frac{W}{N_{PE}} + 24 \quad (5.10)$$

Assuming that the width of the image is twice the number of processors, this reduces to Equation 5.12.

$$N_{CYCLES} = \frac{W \times H}{2 \times N_{PE}} \times (15.5 \times \frac{W}{N_{PE}} + 24) \quad (5.11)$$

$$N_{CYCLES} = H \times 55 \quad (5.12)$$

This represents a speed-up of 58.3% for the dilate operator when compared against Equation 4.14. Using a different dilate algorithm, like the one described in Section 4.4, the modified architecture would experience similar performance improvements as the original architecture.

### 5.3.5 Results

The modifications made to the PE resulted in area reductions ranging from 10.6% to 14.8% and system speed-ups ranging from 0% to 6.7%. The best improvements were from the most recent FPGA studied, the Virtex II Pro, while the worst were from the Virtex E FPGA. The reason that both area and delay results were worse for the Virtex E FPGA is due to the lack of optimized multiplier cores. More of the PE area is consumed by the synthesized multiplier so the decreased area from routing does not have as large of a relative impact as it does for the Virtex II Pro. In fact, the number of LUTs that are saved is equal for both platforms.

Since the Virtex E FPGA requires more LUTs, the critical path does not change significantly so there is no performance increase for the delay in this implementation. For the Virtex II Pro, however, the critical paths are different and changing the routing and size of the PE has a direct impact on the critical path and results in a 6.7% delay savings. Results for an ASIC implementation using $0.18\mu m$ standard cells landed in between the results for the two FPGA implementations.

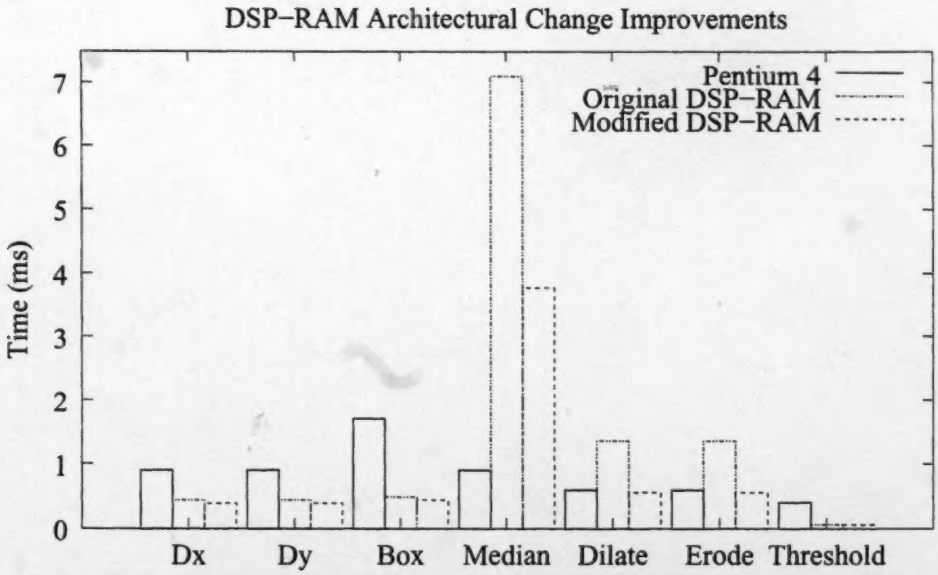



Figure 5.7: Performance improvements over the original DSP-RAM architecture

Figure 5.7 shows the performance improvements over both the original DSP-RAM architecture and over the Pentium 4 for each of the individual algorithms. This shows that for all the algorithms, except thresholding which remained unchanged, performance improvements were realized over the original DSP-RAM architecture.

As anticipated, the dilate, erode and median filters experienced significant improvements (greater than 50%) from the modifications. Sobel dx, and dy along with the box filter show a small improvement due to the 25% improvement in the communication portion of the algorithm. These results can now be used to create Table 5.2, (analogous to Table 4.2), which shows the speed-up of a modified DSP-RAM versus the Pentium 4. With the modified processor, six of the seven algorithms studied show better performance than the Pentium 4. In fact, if the dilate and erode algorithms were to be redesigned as specified in Section 4.4, all six algorithms would be more than twice as fast as the Pentium 4.

The median filter results, while drastically improved, are still not competitive with the results obtained on the Pentium 4. This is due to the large number of instructions that are required for this operation. By using a better sorting algorithm,

Table 5.2: Modified DSP-RAM Image Processing Results (512×512 Image)

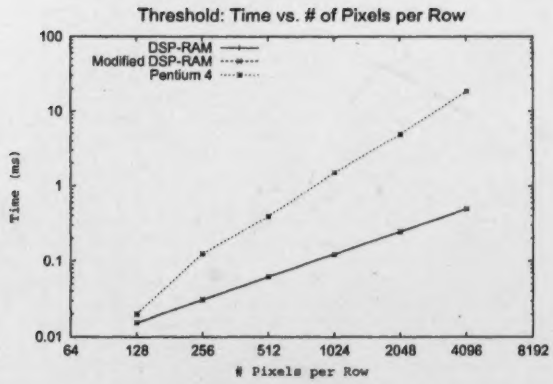
| **Kernel** | **Pentium 4** (1.6 GHz) | **Modified DSP-RAM** (50 MHz) | **Speed-up** |
|---|---|---|---|
| Threshold | 0.4 ms | 0.06 ms | 6.67 |
| Sobel dx | 0.9 ms | 0.39 ms | 2.31 |
| Sobel dy | 0.9 ms | 0.39 ms | 2.31 |
| Dilate | 0.6 ms | 0.56 ms | 1.07 |
| Erode | 0.6 ms | 0.56 ms | 1.07 |
| Box Filter | 1.7 ms | 0.44 ms | 3.86 |
| Median Filter | 0.9 ms | 3.76 ms | 0.24 |

it may be possible to reduce the number of cycles required by this algorithm so that is more competitive against the Pentium 4.

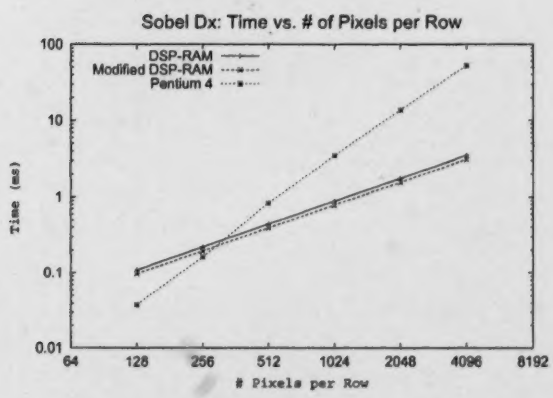
Figures 5.8(a)-5.8(c) show the results for each of the algorithms as the size of the image varies. As before, the threshold operation outperforms the Pentium 4 for all the image sizes shown in Figure 5.8(a). The Sobel filter now matches the performance of the Pentium 4 at around 300 pixels per row, which is slightly smaller than original. The dilate kernel now starts outperforming the Pentium 4 at around 200 pixels per row, which is about 1100 pixels per row earlier than the previous design. The 256 pixel per row point is slightly above the would-be linear curve for the Pentium 4's performance. Since the sequential algorithms time was computed using an average over 10,000 runs, it is unlikely that this is due to machine loading and is more likely due to cache size and utilization with an image of that particular size. However, even if this point were to lie on the linear performance region, the dilate operator would still show a vast improvement over the previous algorithm.

Figure 5.9 shows the result for the median algorithm as the size of the image varies. Even though the median filter is not competitive with the Pentium 4 for the 512 × 512 image size, Figure 5.9 shows that DSP-RAM will start to out-perform the Pentium 4 for images around 2048 pixels and higher.

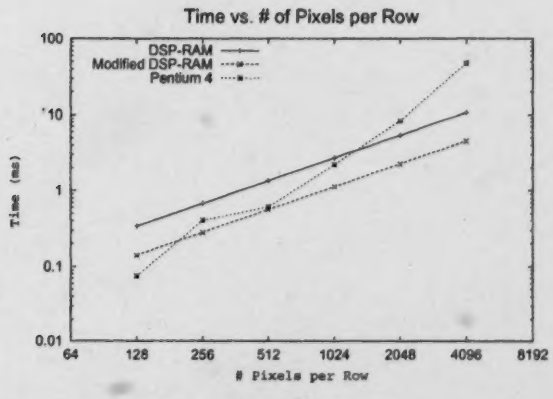
Figure 5.10(a) shows the well-known Lena image used by many image processing researchers. This image was processed by the DSP-RAM architecture for each of the algorithms studied. Figures 5.10(b)-5.10(d) show the resulting images for the three algorithms that were discussed in greater detail in this thesis.




(a) Threshold




(b) Sobel dx




(c) Dilate

Figure 5.8: Computation time versus image size

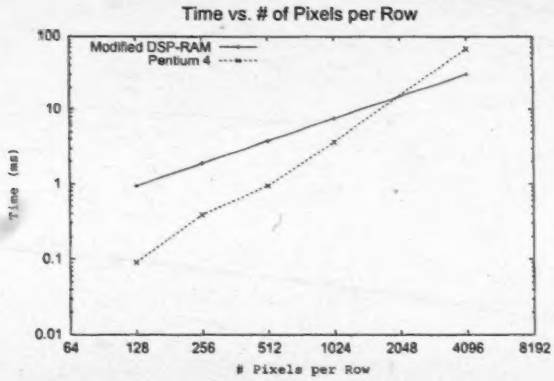
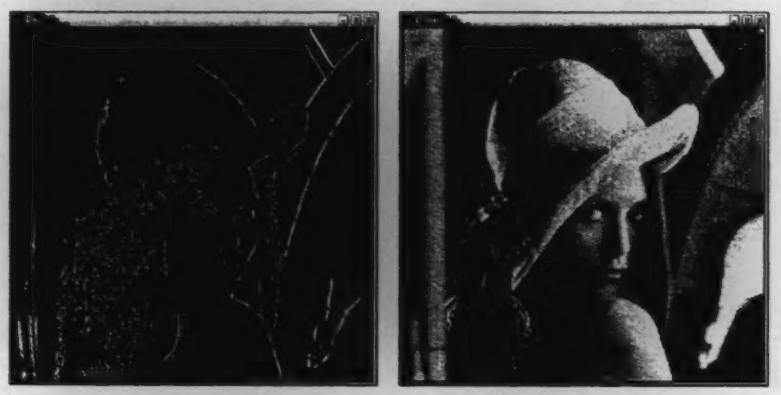



Figure 5.9: Median filter performance for varying image sizes

(a) Original

(b) Threshold



(c) Sobel dx

(d) Dilate

Figure 5.10: DSP-RAM generated images

# Chapter 6

# Future Research Directions

During the course of this research project, we identified four possible research directions in the DSP-RAM area. The first research direction is further architectural analysis within a different signal processing context. For example, while this thesis focused on some fundamental image processing algorithms, other work could study the DSP-RAM architecture in the context of more elaborate computer vision, encryption and security protocols, or communication algorithms. Trying to keep a processor general-purpose, while simultaneously attempting to tune it to many different algorithms is a challenging task.

Next, there is an issue of accuracy. Most of today's DSP families support both 16-bit and 32-bit fixed-point processing as well as a floating-point variant. DSP-RAM could follow this trend set by DSP manufacturers and also maintain different families of processors. All of the DSP-RAM variants could utilize the same programming model, memory structure and network communication, which would ease the burden on the algorithm developer.

Moving from a 16-bit processor to a 32-bit processor would increase the accuracy of fixed-point arithmetic, which can become problematic in 16-bit DSPs when numbers smaller than $\frac{1}{2^{15}}$ must be represented. A 32-bit processor is much larger than a 16-bit one, since the MAC unit and shifter, the two largest components from Table 3.2, must be increased in size. This would seem to go against the benefits of SIMD computing for 8-bit or 16-bit operation. However, the 32-bit PE could be designed so that it utilizes a packed-register much like those of the commercial SIMD extensions studied in Section 2.1. In this way, a single 32-bit DSP-RAM

would be capable of performing in a single cycle four 8-bit MAC operations, two 16-bit MACs, or one 32-bit MAC. In essence, each processing element is a small-scale SIMD array! While the MAC unit is easily modified to handle this mode of operation, special consideration would have to be applied to the enable stack. For instance, four comparisons could easily be done in parallel, but how does one choose which of the results to push on the stack?

Solving this problem, along with others involved in this design, could also be applied directly to the the 16-bit DSP-RAM. If the 16-bit processor were capable of performing two simultaneous 8-bit operations, it is possible that the performance of the image processing algorithms would be close to double, and the byte select circuit could be entirely removed.

While using a 32-bit integer processor would help resolve accuracy issues in DSP algorithms, it would not remove the software complexity associated with fixed-point arithmetic. In fact, the necessary scaling of data with many operations requires additional cycles to maintain the accuracy demanded by the application. Creating a floating-point processing element would be an interesting extension to the DSP-RAM family. A floating-point processor would use less parallelism because of the necessary size of a floating-point MAC unit, but should still be able to compete against typical desktop processors in the floating-point realm. Like most DSP vendors, the floating-point version would be an addition to a family of processors, not a replacement. Using a synthesizable model, the DSP-RAM family could be synthesized as a 16 or 32-bit fixed-point version, as well as a floating-point variant. These could be sold as either ASICs or intellectual property (IP) and integrated into an FPGA along with other circuits.

Next, a solution for the structural hazard that exists in the ALU pipeline (Figure 3.7 could be found. One solution is to convert the ALU to a register-register architecture, or a load-store architecture [17], shown in Figure 6.1. The principle behind this architecture is to utilize a bank of registers, called a register file, as operands to the ALU. Memory access consists of loading a register with memory, or storing a register's contents to memory. These operations are only permitted at one stage of the pipeline, removing the structural hazard.

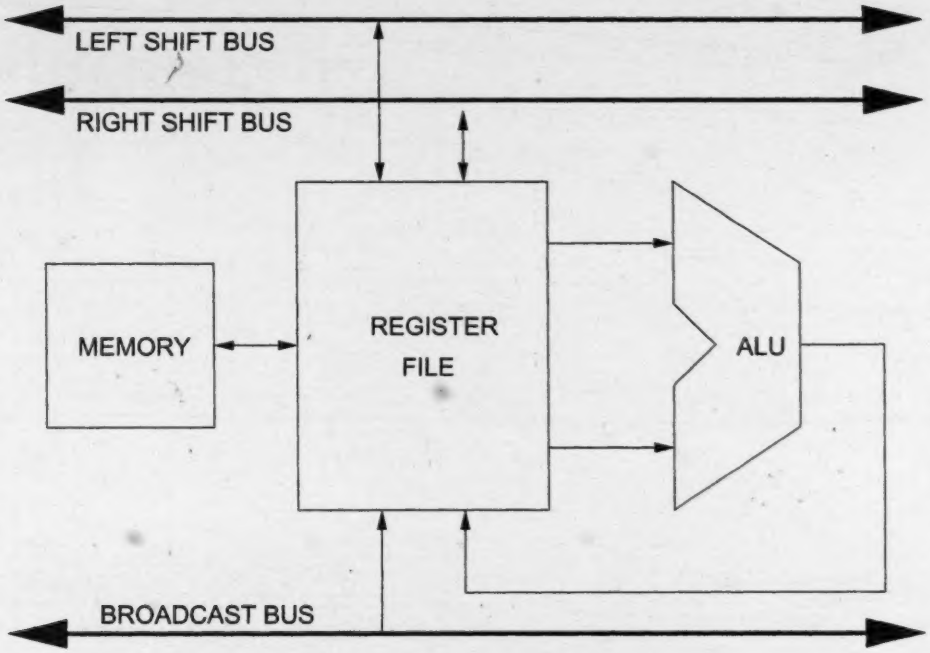



Figure 6.1: Register-register architecture

In fact, the current architecture contains four registers already. These dedicated registers would be converted to general-purpose registers and additional registers would need to be added. The routing complexity would be eliminated by multi-porting the register file. Since the addition of extra ports directly increases the size of the register file, the number would have to be optimized along with area considerations.

This architecture has the additional benefit of being able to use a single memory bank. This decreases the complexity of both the memory controller and the software algorithms. A register file approach is also more readily adapted to an embedded DRAM process. Like the VIRAM archicture described in Section 2.2.3, the register file could act as a small cache for the DRAM and help to amortorize the cost of the higher latency memory core. Use of DRAM would allow for a much greater density over that of an SRAM core.

The final area of research concerns the DSP-RAM controller. The controller used in this work was a test controller that did not operate at the system frequency.

This controller is described in more detail in Appendix A. The test controller provided a method of programming and reading memory and register contents, as well as sending control words to the SIMD array. However, the test controller could only be operated in single-step mode by the host processor. While this allowed us to verify the hardware design, and test algorithms on the development system, it did not allow the system to be clocked at the system frequency. It also does not allow DSP-RAM to be used in a stand-alone system. The host processor is required to send the control words to the SIMD array. A better controller would support control structures such as branching and comparison at the array level. The controller could also fetch and possibly decode instructions directly from memory, allowing it to operate in a stand-alone environment. A more detailed look at SIMD controllers is given in Noah Akililu's M.Sc. thesis [2]. A DSP-RAM specific controller is described in Billie Kwan's M.Eng. report [22].

# Chapter 7

# Discussion and Conclusions

This research presents DSP-RAM as an attractive solution to the twin goals of embedded image processing: high-performance computing and low-power consumption. DSP-RAM is a SIMD archicture that integrates several DSPs with an SRAM memory core. This union increases the usable bandwidth by exposing the data available at the sense amplifiers of the memory to the DSP processors. This allows DSP-RAM to compete against high-performance microprocessors at a much lower clock frequency and with reduced power consumption.

In order to form a context for this research, we presented several parallel architectures in use in today's high-performance microprocessors, along with survey of parallel architectures developed in the research field. Next the image processing benchmarks that we used were described. Sources of power dissipation in digital CMOS design was also given in order to back our claims of DSP-RAM's low power consumption. Chapter two was concluded by describing the development platform—the ARM Integrator-based Rapid-Prototyping Platform.

The major contributions made by this thesis include:

- Creation of a synthesizable VHDL model of the DSP-RAM architecture
- Detailed area and delay analysis for each PE component
- Location of the three primary critical paths in the PE
- Implementation of 7 image processing algorithms
- Proposal and evaluation of two architectural enhancements

- Performance comparison against an Intel Pentium 4 processor

First, a synthesizable VHDL model was created. Previously, only the C++ emulator and a non-synthesizable VHDL model existed. A synthesizable model allowed the architecture to be analyzed with respect to component size and timing (Table 3.2. Analyzing the timing of the processor and locating the three critical paths (Table 3.3), revealed sub-optimal pipeline balancing for an FPGA with internal multiplier cells and also for an ASIC implementation. The model was also parameterized to allow for algorithm-independent modifications such as reducing the number of overhead bits in the MAC, something that was not available in the previous VHDL model. This allowed us to synthesize a minimum-sized PE, which provided an area savings of 36% (Table 3.4.

Chapter 4 went on to explain in detail the design and results for three of the algorithms studied. These algorithms were compared against the same algorithms executed on a 1.6-GHz Pentium 4 using the SSE2 extensions. Assuming a modest 50-MHz DSP-RAM, four of the seven algorithms experienced speed-ups of more that two times. Two of the algorithms, dilate and erode, were only half as fast, and the median filter experienced the worst performance measuring only 13% that of the Pentium 4 (Table 4.2).

The processor was then analyzed with two main goals in mind. The first goal was to decrease the complexity and size of the process without increasing the performance of the algorithms. The second goal was to increase the performance of the three sub-par algorithms. By reducing the interconnect between PEs and efficiently pipelining the transfer into the memory banks, the area consumed by the interconnect wiring was reduced to a third. Only specialized algorithms, like an FIR filter that shifts data directly into the accumulator, experienced a performance hit. Next, using existing hardware resources, an ALU by-pass path was added into the PE that allowed data to short-circuit the high-latency ALU pipeline and write directly back to memory. These modifications reduced the area by 10 to 15% depending on what architecture was synthesized (Table 5.1). While the speed of the median filter only increased to 24% of that of the Pentium 4 with these modifications, dilate and erode out-performed the Pentium 4 by increasing from 44% to 107%. The other

algorithms experienced either no increase (threshold) or only a small improvement.

In summary, the quantitative analysis of the DSP-RAM processing architecture provides a strong base for further development on this platform. The VHDL model is fully parameterized, allowing for rapid experimentation with different architectures. Providing synthesis results for an ASIC process using $0.18 - \mu m$ standard cells offers a low-power alternative to conventional digital signal processors, without sacrificing the performance seen on a desktop. While a low-power solution requires the use of an ASIC process, we have shown that an FPGA implementation can also compete with a high-performance computer. Further, the parallelism inherent in FPGAs can be immediately exploited by the DSP-RAM SIMD architecture by re-synthesizing the design with more processors. This allows DSP-RAM to benefit from not only the increase in parallelism, but also the increase in performance from the next generations of FPGAs.

# Bibliography


[1] Yoshiharu Aimoto, Tohru Kimura, Yoshikazu Yabe, Hideki Heiuchi, Youetsu Nakazawa, Masato Motomura, Takuya Koga, Yoshihiro Fujita, Masayuki Hamada, Takaho Tanigawa, Hajime Nobusawa, and Kuniaki Koyama. A 7.68 GIPS 3.84GB/s 1W Parallel Image-Processing RAM Integrating a 16Mb DRAM and 128 Processors. In *IEEE International Solid-State Circuits Conference*, pages 372–373,476. IEEE, February 1996.

[2] Noah Aklilu. Integrating Computational RAM (C•RAM). Master's thesis, University of Alberta, 2001.

[3] *AltiVec®Technology Programming Environments Manual*, February 2002.

[4] ARM. *AMBA™ Specification (Rev. 2)*, 1999.

[5] ARM. *Integrator™/CM7TDMI: User Guide*, 1999.

[6] ARM. *Integrator™/LM-XCV600E+: User Guide*, 2000.

[7] ARM. *Integrator™/AP: User Guide*, 2001.

[8] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19:23–29, July-August 1999.

[9] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power CMOS Digital Design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[10] CMC Rapid Proto-Typing Platform, 2003.

[11] K. Diefendoriff. PC Processor Microarchitecture. *Microdesign Resources, Microprocessor Report*, pages 1–7, July 1999.

[12] Steve J. Dillen and Bruce F. Cockburn. Parallel Filtering and Thresholding of Images on the SIMD DSP-RAM Architecture. *IEEE Canadian Conference of Electrical and Computer Engineering*, May 2002.

[13] Duncan G. Elliott. *Computational RAM: A Memory-SIMD Hybrid*. PhD thesis, University of Toronto, 1998.

[14] Sam Fuller. Motorola's AltiVec™Technology. Technical Report ALTIVECWP/D, Motorola Inc. Semiconductor Product Sector, 1998.

[15] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in Memory: The Terasys Massively Parallel PIM Array. *Computer*, 28(4):23–31, April 1995.

[16] Greg Hager. Efficient Region Tracking with Parametric models of geometry and illumination. *IEEE Transactions on PAMI*, 1997.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2003.

[18] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Digest of Technical Papers, IEEE Symposium on Low-Power Electronics*, pages 8–11, 1994.

[19] *IA-32 Intel Architecture Software Developer's Manual*, 2002.

[20] Christoforos Kozyrakis. *Scalable Vector Media-processors for Embedded Systems*. PhD thesis, University of California - Berkeley, 2002.

[21] Christoforos Kozyrakis, David Judd, Joseph Gebis, Samuel Williams, David Patterson, and Katherine Yelick. Hardware/Compiler Codevelopment for an Embedded Media Processor. *Proceedings of the IEEE*, 89(11), November 2001.

[22] Bill S.-H. Kwan. DSP-RAM. Technical report, University of Alberta, 2002.

[23] *Pentium®Processor Family Developer's Manual*, 1997.

[24] M.D. Powell, S.-H. Yang, B. Falsafi, and K. Roy. Gated-Vdd: A circuit technique to reduce leakage in cache memories. In *International Symposium Low-Power Electronics Design (ISPLED)*, pages 90–95, July 2000.

[25] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and Vijaykumar T. N. Reducing Leakage in a High-Performance Deep-Submicron Instruction Cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):77–89, February 2001.

[26] S. Przybylski. New DRAM Technologies: A Comprehensive Analysis of the New Architectures. *MicroDesign Resources*, 1994.

[27] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing Streaming SIMD Extensions on the Pentium III Processor. *IEEE Micro*, pages 47–57, July-August 2000.

[28] John C. Russ. *The Image Processing Handbook*. CRC Press, 4th edition, 2002.

[29] Nat Seshan. High VelociTI Processing. *IEEE Signal Processing Magazine*, pages 86–117, March 1998.

[30] Linda G. Shapiro and George C. Stockman. *Computer Vision*. Prentice Hall, 2001.

[31] Nathan Slingerland and Alan Jay Smith. Measuring the Performance of Multimedia Instruction Sets. *IEEE Transactions on Computers*, 51(11):1317–1332, November 2002.

[32] William Stallings. *Computer Organization and Architecture: Designing for Performance*. Prentice Hall, 5th edition, 2000.

[33] Sun Microsystems, Inc. *The VIS™ Instruction Set*, June 2002.

[34] Synopsys. *DesignWare Foundation Library Databook*, August 2001.

[35] *TMS320C6000 CPU and Instruction Set Reference Guide*, October 2000.

[36] TSMC 0.18$\mu$ process technology. http://www.tsmc.com.tw/english/technology/.

[37] *UltraSPARC$^{TM}$-IIi User's Manual*, 1997.

[38] John P. Uyemura. *Introduction to VLSI Circuits and Systems*. John Wiley & Sons, 2002.

[39] *Virtex-E 1.8V Field Programmable Gate Arrays: Detailed Description*, December 2002.

[40] *Virtex-II Pro$^{TM}$ Platform FPGAs: Functional Description*, January 2003.

[41] *Virtex-E 1.8V Field Programmable Gate Arrays: Functional Description*, November 2001.

[42] Zixiong Wang, Bruce F. Cockburn, Duncan G. Elliott, and Witold A. Krzymien. DSP-RAM: A Logic-Enhanced Memory Architecture for Communication Signal Processing. *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 475–478, Aug 1999.

[43] Nobuyuki Yamashita, Tohru Kimura, Yoshihiro Fujita, Yoshiharu Aimoto, Takashi Manabe, Shin'ichiro Okazaki, Kazuyuki Nakamura, and Masakazu Yamashina. A 3.84 GIPS Integrated Memory Array Processor with 64 Processing Elements and a 2-Mb SRAM. *IEEE Journal of Solid-state Circuits*, 29(11):1336–1343, November 1994.

[44] Gary K. Yeap. *Practical Low-power Digital VLSI Design*. Kluwer Academic Publishers, 1998.

# Appendix A

# DSP-RAM Controller Implementation

The DSP-RAM Controller was implemented as an AMBA Peripheral Bus (APB) module so that it could easily be integrated into the rapid prototyping platform. The controller maintains fourteen 32-bit registers that can be addressed from the ARM processor on the Integrator core module. This allows us to develop DSP-RAM algorithms using the C language, compile them on the ARM processor and send them to the DSP-RAM executing on the logic module.

## A.1 Register description

Table A.1 provides the addresses for all the DSP-RAM controller registers.

The first register that will be discussed is the control register. The control register is used to reset and clock the DSP-RAM system. While it is a 32-bit register, only 2 of the bits are used, allowing for an additional 30 control bits to be used for a more elaborate controller. Bit 0 of the control register is used as the system clock and bit 15 is used as reset, as shown by Figure A.1. When the clock bit transitions

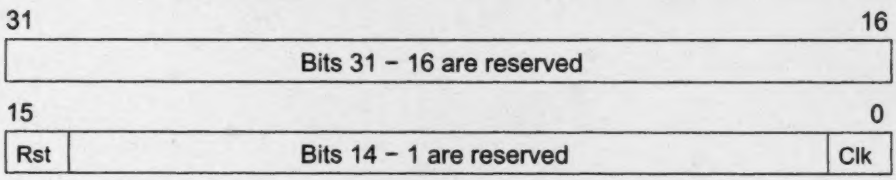



Figure A.1: Control register

Table A.1: DSP-RAM Controller Registers

| Address | Register | Description |
|---|---|---|
| 0xC0000000 | Address A | Bank A memory address |
| 0xC0000004 | Address B | Bank B memory address |
| 0xC0000008 | Data A | Data to/from memory bank A |
| 0xC000000C | Data B | Data to/from memory bank B |
| 0xC0000010 | Control Word LSW | Least significant 32 bits of the control word |
| 0xC0000014 | Control Word MSW | Most significant 32 bits of the control word |
| 0xC0000018 | Left Bus 0 | Bits 15-0 of the left shift bus |
| 0xC000001C | Left Bus 1 | Bits 31-16 of the left shift bus |
| 0xC0000020 | Left Bus 2 | Bits 47-32 of the left shift bus |
| 0xC0000024 | Right Bus 0 | Bits 15-0 of the right shift bus |
| 0xC0000028 | Right Bus 1 | Bits 31-16 of the right shift bus |
| 0xC000002C | Right Bus 2 | Bits 47-32 of the right shift bus |
| 0xC0000030 | Broadcast Bus | Broadcast bus |
| 0xC0000034 | Control Register | Control bits for the controller |

from a logic '0' to a logic '1', this generates a rising edge. Similarly, when the clock bit transitiitions from a logic '1' to a logic '0', a falling edge is produced.

The address A, address B and data A and data B registers are used to access the memory banks. Data A (B) is the 16-bit data port on memory bank A (B). This can be used to read or write to the memory bank as a conventional SRAM. The address registers are used to program which address is being accessed. These must be programmed before accessing the memory banks. The address registers are also used to program the memory address for SIMD instructions that require the use of the memory banks.

The broadcast bus register is connected to the broadcast bus of the DSP-RAM. This register can be programmed with a specific broadcast value before the DSP-RAM system is clocked. The left and right bus registers work the same way. The three registers are connected to the 48-bit shift bus on the DSP-RAM. If a DSP-RAM is synthesized with only a 16-bit shift bus, then only the left bus 0 and right bus 0 registers are used. The DSP-RAM controller can also extract values from the shift bus using these registers by reading the addresses instead of writing to them.

Finally, the control word registers are used to control the DSP-RAM processing element. These two registers contain 50 control bits, shown in Table A.2, leaving

13 available for expansion. Most of these are self-explanatory when referencing Figure 3.3. The signals that need further explanation are the shift value signals. The shift value uses three bits to represent the 8 possible shift values. These distances are set through generic parameters in the VHDL model. When a shift value of '000' is programmed, it uses the value assigned to the *sv0* generic parameter. When a shift value of '100' is programmed, it uses the value assigned to the *sv4* generic parameter. If a DSP-RAM is synthesized with only four shift values, then bit 3 is ignored. Next, the shift mode controls how the shifter will operate. These codes are summarized in Table A.3.

Next the stack operation mode is defined in Table A.4.

Finally, bit 49 to bit 44 are used by the controller to determine what operations are being done on the entire DSP-RAM system. Bits 44 and 45 control read and write access on the memory ports, both for sequential memory operations and SIMD memory operations. Likewise, bits 46 and 47 operate the write enable signals for the memories. For the SIMD memory operations, these signals are ANDed with the RAM enable signal from the stack in the processing elements to selectively enable the RAMs. For sequential operations, bits 48 and 49 are used to selectively route either the SIMD enable signals or the unmodified ram enable signals. This means that even if a PE has a memory bank disabled, a sequential write can still be committed to memory.

## A.2 Example programs

Listing A.1 provides an example of how to execute a sequential memory load into DSP-RAM memory bank A. The first two lines write the control word. Since this is a sequential operation, all the bits used to control the PE can be disabled. The MSW is programmed to set bits 46 and 48. Bit 46 indicates a write on bank A, and bit 48 tells the controller that it is a sequential access. Next the address and data are programmed into the DSP-RAM controller. Finally, the DSP-RAM system is clocked using the control register.

Table A.2: Control Word Register Definition

| Bit(s) | Description |
|---|---|
| 0 | Adder Enable |
| 3-1 | Shift Value |
| 5-4 | Shift Mode |
| 6 | ALU Enable |
| 7 | Input A mux select |
| 8 | Byte select |
| 9 | Byte select enable |
| 10 | Input B mux select |
| 11 | Left or right shift bus mux select |
| 12 | Accumulator 0 mux select |
| 13 | Accumulator 1 mux select |
| 14 | Accumulator 2 mux select |
| 15 | Accumulator reset |
| 16 | Reserved |
| 19-17 | Accumulator 2-0 Enables |
| 21-20 | Broadcast bus mux select |
| 22 | Broadcast bus enable |
| 25-23 | Stack mux select |
| 27-26 | Stack operation mode |
| 28 | Arbiter Enable |
| 30-29 | Data bus mux select |
| 31 | Data bus A driver enable |
| 32 | Data bus B driver enable |
| 33 | Data bus left shift bus driver enable |
| 34 | Data bus right shift bus driver enable |
| 35 | Left shift bus direction |
| 36 | Right shift bus direction |
| 38-37 | Shift register mux select |
| 39 | Shift register enable |
| 40 | Shift register left shift bus driver enable |
| 41 | Shift register right shift bus driver enable |
| 43-42 | Reserved |
| 44 | Bank A Read/Write |
| 45 | Bank B Read/Write |
| 46 | Bank A write enable |
| 47 | Bank B write enable |
| 48 | Sequential access to bank A requested |
| 49 | Sequential access to bank B requested |

Table A.3: Shift Modes

| Code | Meaning |
|---|---|
| 00 | No shift, shifter is disabled |
| 01 | Logical shift left |
| 10 | Logical shift right |
| 11 | Arithmetic shift right |

Table A.4: Shift Modes

| Code | Meaning |
|---|---|
| 00 | NOP, stack is disabled |
| 01 | PUSH, push a value onto the stack |
| 10 | POP, pop a value from the stack |
| 11 | TOGGLE, toggle the current stack value |

Listing A.2 provides a more detailed example of an FIR filter on DSP-RAM. This listing is can be used as a reference for programming more algorithms and will not be explained any further.

Listing A.2: DSP-RAM FIR Filter

```
/*****************************************************************************
 * Copyright ARM Limited 1999.  All rights reserved.
 *****************************************************************************/
/*****************************************************************************
 *
 *      $Id: logic.c,v 1.2 1999/11/22 10:58:54 mwelsh Exp $
 *
 *****************************************************************************/


#include <stdio.h>
#include "logic.h"
#include "platform.h"

// #define DEBUG_MODE
#define NUM_PEs         16
#define FILTER_LENGTH 4

int firInput( int data )
{
```

Listing A.1: Load DSP-RAM memory

```
void writeMemoryA( int address, short data )
{

  /* This function requires that the DSP-RAM clock has previously been
   * programmed low
   */
  word_write( LM_CONTROLWORD_LSW, 0x00000000 ); /* Sequential operation  */
  word_write( LM_CONTROLWORD_MSW, 0x00014000 ); /* Write memory A        */

  word_write( LM_ADDRESS_A, address );          /* AddressA = 0          */
  word_write( LM_DATA_A, data );                /* DataIn = 2            */
  word_write( LM_CONTROLREGISTER, 0x8001 );     /* Clock = 1, nReset = 1 */
  word_write( LM_CONTROLREGISTER, 0x8000 );     /* Clock = 0, nReset = 1 */

#ifdef DEBUG_MODE

  printf( "Address_%d_=_%d\n", address, word_read( LM_DATA_A ) );

#endif

} /* writeMemoryA */
```

```
  int firOutput = 0;

  /* First FIR instruction */
  word_write( LM_BROADCASTBUS, data );              /* Broadcast the data  */

#ifdef DEBUG_MODE
  printf( "BB_=_%d\n", word_read( LM_BROADCASTBUS ) );
#endif

  word_write( LM_CONTROLWORD_LSW, 0x0008AA01 );  /* First Op Code - LSW */
  word_write( LM_CONTROLWORD_MSW, 0x00007BC0 );  /*                - MSW */

#ifdef DEBUG_MODE
  printf( "CW_=_%08x%08x\n", word_read( LM_CONTROLWORD_MSW ),
                             word_read( LM_CONTROLWORD_LSW ) );
#endif

  word_write( LM_CONTROLREGISTER, 0x00008001 );  /* Clock DSP-RAM       */

#ifdef DEBUG_MODE
  printf( "CR_=_%04x\n", word_read( LM_CONTROLREGISTER ) );
#endif

  word_write( LM_CONTROLREGISTER, 0x00008000 );

#ifdef DEBUG_MODE
  printf( "CR_=_%04x\n", word_read( LM_CONTROLREGISTER ) );
#endif

/* Second FIR instruction */
  word_write( LM_CONTROLWORD_LSW, 0x00080201 );  /* First Op Code - LSW */
  word_write( LM_CONTROLWORD_MSW, 0x000073D8 );  /*                - MSW */

#ifdef DEBUG_MODE
  printf( "CW_=_%08x%08x\n", word_read( LM_CONTROLWORD_MSW ),
                             word_read( LM_CONTROLWORD_LSW ) );
#endif

  firOutput = word_read( LM_RIGHTBUS_0 );        /* Read filter output  */
  word_write( LM_CONTROLREGISTER, 0x00008001 );  /* Clock DSP-RAM       */
  word_write( LM_CONTROLREGISTER, 0x00008000 );

  return firOutput;

} /* firInput */

void writeMemoryA( int address, short data )
{

  word_write( LM_ADDRESS_A, address );        /* AddressA = 0           */
  word_write( LM_DATA_A, data );              /* DataIn = 2             */
  word_write( LM_CONTROLREGISTER, 0x8001 );   /* Clock = 1, nReset = 1 */
  word_write( LM_CONTROLREGISTER, 0x8000 );   /* Clock = 0, nReset = 1 */

#ifdef DEBUG_MODE

  printf( "Address_%d_=_%d\n", address, word_read( LM_DATA_A ) );

#endif

} /* writeMemoryA */

int main(void)
{

  int i,j, SwitchValue;

  printf( "Testing_DSP-RAM_with_FIR_Filter.\n" );

  /* Reset the DSP-RAM */
  word_write( LM_CONTROLREGISTER, 0x0001 );   /* Clock = 1, nReset = 0 */
  word_write( LM_CONTROLREGISTER, 0x8000 );   /* Clock = 0, nReset = 1 */
  word_write( LM_CONTROLREGISTER, 0x8001 );   /* Clock = 1, nReset = 1 */
  word_write( LM_CONTROLREGISTER, 0x8000 );   /* Clock = 0, nReset = 1 */

  /* Program the control register for sequential writes */
  word_write( LM_CONTROLWORD_LSW, 0x00000000 );
  word_write( LM_CONTROLWORD_MSW, 0x00014000 );

  /* Load the filter coefficients */

  /* Zero the first memory addresses */
  for ( i = 0; i < (NUM_PEs - FILTER_LENGTH); i++ )
  {

    writeMemoryA( i, 0 );

  } /* for */

  /* Write the filter coefficients */
  i = (NUM_PEs - FILTER_LENGTH);
```

```
    writeMemoryA( i++, 2  );
    writeMemoryA( i++, 10 );
    writeMemoryA( i++, 5  );
    writeMemoryA( i++, 8  );

    /* Send in the input stream */
    printf( "FIR_Output_=_%d\n", firInput( 0 ) );
    printf( "FIR_Output_=_%d\n", firInput( 2 ) );
    printf( "FIR_Output_=_%d\n", firInput( 0 ) );
    printf( "FIR_Output_=_%d\n", firInput( 1 ) );
    printf( "FIR_Output_=_%d\n", firInput( 0 ) );
    printf( "FIR_Output_=_%d\n", firInput( 0 ) );
    printf( "FIR_Output_=_%d\n", firInput( 0 ) );
    printf( "FIR_Output_=_%d\n", firInput( 0 ) );
    printf( "FIR_Output_=_%d\n", firInput( 0 ) );
    printf( "FIR_Output_=_%d\n", firInput( 0 ) );
    printf( "FIR_Output_=_%d\n", firInput( 0 ) );
    printf( "FIR_Output_=_%d\n", firInput( 0 ) );

    printf( "\n" );

    return 0;

} /* main */
```

# Appendix B

# DSP-RAM C++ Emulator Source Code

## B.1 DSP-RAM Class

Listing B.1: DSP-RAM Class Definition

```
//
//  DSPRAMClass Class Declaration
//
//  Version 1.0   February 10, 2000
//          1.1   February 11, 2000
//          1.2   February 16, 2000
//          1.3   February 20, 2000
//  Version 1.4   March    24, 2000   New RAM EN stack; shifter
//                                    mult-add-shift-accload takes 4 clock cycles
//  Version 2.0   October  26, 2000
//          2.1   November 21, 2000   Added by-pass path around shift bus register
//
// $Id: dspramClass.h,v 1.2 2002/08/06 16:49:52 dillen Exp dillen $
//
// $Log: dspramClass.h,v $
// Revision 1.2  2002/08/06 16:49:52  dillen
// Added 16-Bit Shift bus option
//
// Revision 1.1  2002/08/01 16:10:48  dillen
// Initial revision
//

#ifndef DSPRAMCLASS_H
#define DSPRAMCLASS_H

#include <iostream>
#include "integerClasses.h"
#include "shiftValueType.h"

#define NUM_PEs 64
#define NUM_RAM_BANKs 2
#define RAM_BANK_SIZE 1024
#define STACK_SIZE 10
#define BANK_A 0
#define BANK_B 1
#define ACC_0 0
#define ACC_1 1
#define ACC_2 2

typedef enum
{

  WORD_0                    = 0,
  WORD_1                    = 1,
  WORD_2                    = 2,
  NUMBER_OF_WORD_POSITIONS  = 3

} WordPosition;

#define SHIFTBUS_16
#define ENHANCED_ROUTING

class DSPRAMClass {
public:
```

```
    DSPRAMClass( int numPEs = NUM_PEs,
                 int ramSize = RAM_BANK_SIZE,
                 int stackSize = STACK_SIZE );  // DSP-RAM constructor

    // DSP-RAM destructor
    ~DSPRAMClass();

    // Memory allocation functions.  Return value is the base address.
    int intAlloc( int bank_num );     // Used to allocated one 16-bit parallel variable
    int intVectorAlloc( int bank_num, int vectorLength );

    // Microprocessor port write and read operations
    void write( Int16 data, int pe_num, int bank_num, int address );
    Int16 read( int pe_num, int bank_num, int address );

    // Shift bus operations
#ifdef SHIFTBUS_16
    void injectIntoLeft( Int16 injectedvalue );
    void injectIntoRight( Int16 injectedValue );
    Int16 extractFromLeft();
    Int16 extractFromRight();
    void rightBusFromAcc( WordPosition position = WORD_0 );
    void leftBusFromAcc( WordPosition position = WORD_0 );

#ifdef ENHANCED_ROUTING

    void saveInputA( int bankNumber, int address );
    void rightBusFromInputA( void );
    void leftBusFromInputA( void );

#endif

#else
    void injectIntoLeft( Int48 injectedvalue );
    void injectIntoRight( Int48 injectedValue );
    Int48 extractFromLeft();
    Int48 extractFromRight();
    void rightBusFromAcc();
    void leftBusFromAcc();
#endif

    void leftBusFromShReg();
    void rightBusFromShReg();

    // Broadcast bus operations
    void injectOntoBroadcastBus( Int16 injectedValue );
    Int16 extractFromBroadcastBus();

    // Register operations
    void regLoadBroadcastBus();
    void regWriteBroadcastBus();
    void regLoadDataA( int addressA );
    void regLoadDataB( int addressB );
    void regLoadAccWord( int accByteNumber );  // argument is either 2, 1, or 0

    // Accumulator operations
    void accClear();                          // all three accumulator words cleared
    void accLoadShifter();

    // Accumulator word operations
    void acc0LoadShifter();      // added in Version 2.1
    void acc1LoadShifter();
    void acc2LoadShifter();

#ifdef SHIFTBUS_16
    void accLoadFromLeft( WordPosition position = WORD_0 );
    void accLoadFromRight( WordPosition position = WORD_0 );
#else
    void accLoadFromLeft();                   // all three accumulator words loaded
    void accLoadFromRight();                  // all three accumulator words loaded
#endif

    // Shift register operations  (new to version 2.0)
    void shRegClear();
#ifdef SHIFTBUS_16
    void shRegLoadShifter( WordPosition position = WORD_0 );
#else
    void shRegLoadShifter( void );
#endif

    void shRegLoadFromLeft();
    void shRegLoadFromRight();

    // Multiplier input configuration operations (new to version 2.0)
    void multAxB( int addressA, int addressB );
    void multBxB( int addressB );
    void multAxR( int addressA );
    void multBxR( int addressB );

    // Adder control operations  (new to Version 2.1)
    void enableAdder();
    void disableAdder();
```

```
    // Shifter control operations  (new to Version 2.1)
    void setShifter( ShiftValue );

    // Enable masking with the BUS register
    void enableByteSelect( int byte=0 );
    void disableByteSelect();

    // Memory load operations
    void memLoadAccWord( int bank_num, int address, int accByteNumber );

#ifdef ENHANCED_ROUTING

    void memLoadInputA( int bank_num, int address );

#endif


    // Memory enable operations.  The bank_num argument is either 0 or 1.
    void pushEnableAeqB( int addressA, int addressB );
    void pushEnableAeqR( int addressA );
    void pushEnableBeqR( int addressB );
    void pushEnableAltB( int addressA, int addressB );
    void pushEnableAltR( int addressA );
    void pushEnableBltR( int addressB );
    void pushEnableAgtB( int addressA, int addressB );
    void pushEnableAgtR( int addressA );
    void pushEnableBgtR( int addressB );
    void pushEnableAgeB( int addressA, int addressB );
    void pushEnableAgeR( int addressA );
    void pushEnableBusResult(); // Broadcast bus compare
    void toggleEnable();
    void popEnable();   // new in version 2.0
    // void saveEnable( int bank_num, int address );
    // void restoreEnable( int bank_num, int address );
    // void loadEnableLSB();

    // The following two operations are used to group parallel DSP-RAM operations
    void startInstructionDefn( int = 0);  // verifies consistency of microoperations
    void endInstructionDefn( int = 0);    // executes instruction cycle

    // Debug functions
    void dumpAcc( void );

    // Access functions
    const int getCycleCount( void );
    const int getNumberOfPEs( void );
    const int getRamBankSize( void );
    const int getHwStackSize( void );

private:
    Int16 broadcastBusValue;

#ifdef SHIFTBUS_16
    Int16 leftBusValue;
    Int16 rightBusValue;
#else
    Int48 leftBusValue;
    Int48 rightBusValue;
#endif
    bool  leftInjectorsEnabled, rightInjectorsEnabled;
    int   stackPtr;
    bool  pushStackEn, popStackEn, toggleStackEn;
    int   nextAvailableAddressA, nextAvailableAddressB;
    bool  adderEnabled;                    // new in Version 2.1
    ShiftValue currentShiftValue;          // new in Version 2.1
    bool  byteSelectEnabled;               // SJD addition
    int   useByte;                         // SJD addition
    int   cycleCount;
    void  incrCycleCount();
    void  printRegisters();

    // Define the hardware registers
    Int16 * acc0;
    Int16 * acc1;
    Int16 * acc2;
    Int16 * acc0Next;
    Int16 * acc1Next;
    Int16 * acc2Next;

    Int16 * multInput1;
    Int16 * multInput2;
    Int16 * multInput1Next;
    Int16 * multInput2Next;

    Int16 * broadBusReg;
    Int16 * broadBusRegNext;

    Int32 * multMidReg;
    Int32 * multMidRegNext;
    Int32 * multOutput;
    Int32 * multOutputNext;
```

```
#ifdef SHIFTBUS_16
    Int16 * shReg;
    Int16 * shRegNext;
    Int16 * leftShiftBusValue;
    Int16 * rightShiftBusValue;
#else
    Int48 * shReg;
    Int48 * shRegNext;
    Int48 * leftShiftBusValue;
    Int48 * rightShiftBusValue;
#endif

    Int16 *** ram;

    bool *  ramEnNext;
    bool ** ramEnStack;

    int   numberOfPEs;
    int   ramBankSize;
    int   hwStackSize;
};

#endif
```

## Listing B.2: DSP-RAM Class Implementation

```
//
//  DSPRAMClass Class Function Definitions
//
//  Version 1.0   February 16, 2000
//  Version 1.1   March    24, 2000
//          1.2   October  15, 2000    Implemented DSPRAMClass functions
//  Version 2.0   October  26, 2000
//          2.1   November 21, 2000
//
// $Id: dspramClass.cpp,v 1.2 2002/08/06 16:50:35 dillen Exp dillen $
//
// $Log: dspramClass.cpp,v $
// Revision 1.2  2002/08/06 16:50:35  dillen
// Added a 16-Bit shift bus
//
// Revision 1.1  2002/08/01 16:12:59  dillen
// Initial revision
//

#include <assert.h>
#include <stdlib.h>
#include "dspramClass.h"
#include <iomanip>

using std::cout;
using std::cerr;
using std::endl;
using std::setw;
using std::setfill;

DSPRAMClass::DSPRAMClass( int numPEs, int ramSize, int stackSize )
{
    int p, j;

    // Make sure the parameters make sense
    assert( numPEs > 0 );
    assert( ramSize > 0 );
    assert( stackSize > 0 );

    // Initialize hardware variables
    numberOfPEs = numPEs;
    ramBankSize = ramSize;
    hwStackSize = stackSize;

    // Create the hardware registers for each PE
    acc0     = new Int16[numberOfPEs]; assert( acc0     != NULL );
    acc0Next = new Int16[numberOfPEs]; assert( acc0Next != NULL );
    acc1     = new Int16[numberOfPEs]; assert( acc1     != NULL );
    acc1Next = new Int16[numberOfPEs]; assert( acc1Next != NULL );
    acc2     = new Int16[numberOfPEs]; assert( acc2     != NULL );
    acc2Next = new Int16[numberOfPEs]; assert( acc2Next != NULL );

    multInput1     = new Int16[numberOfPEs]; assert( multInput1     != NULL );
    multInput2     = new Int16[numberOfPEs]; assert( multInput2     != NULL );
    multInput1Next = new Int16[numberOfPEs]; assert( multInput1Next != NULL );
    multInput2Next = new Int16[numberOfPEs]; assert( multInput2Next != NULL );
    multMidReg     = new Int32[numberOfPEs]; assert( multMidReg     != NULL );
    multMidRegNext = new Int32[numberOfPEs]; assert( multMidRegNext != NULL );
    multOutput     = new Int32[numberOfPEs]; assert( multOutput     != NULL );
    multOutputNext = new Int32[numberOfPEs]; assert( multOutputNext != NULL );

    broadBusReg    = new Int16[numberOfPEs]; assert( broadBusReg    != NULL);
```

```
    broadBusRegNext = new Int16[numberOfPEs]; assert( broadBusRegNext != NULL);

#ifdef SHIFTBUS_16
    leftShiftBusValue  = new Int16[numberOfPEs]; assert( leftShiftBusValue != NULL);
    rightShiftBusValue = new Int16[numberOfPEs]; assert( rightShiftBusValue!= NULL);

    shReg     = new Int16 [numberOfPEs];  assert( shReg != NULL);
    shRegNext = new Int16 [numberOfPEs];  assert( shRegNext != NULL);

#else
    leftShiftBusValue  = new Int48[numberOfPEs]; assert( leftShiftBusValue != NULL);
    rightShiftBusValue = new Int48[numberOfPEs]; assert( rightShiftBusValue!= NULL);

    shReg     = new Int48 [numberOfPEs];  assert( shReg != NULL);
    shRegNext = new Int48 [numberOfPEs];  assert( shRegNext != NULL);

#endif

    // Create the memory banks and shift register
    ram        = new Int16 *[numberOfPEs]; assert( ram != NULL );
    ramEnNext  = new bool[numberOfPEs];    assert( ramEnNext );
    ramEnStack = new bool *[numberOfPEs];  assert( ramEnStack );


    for (int pe=0; pe<numberOfPEs; pe++)
    {

      ram[pe] = new Int16 *[NUM_RAM_BANKS]; assert( ram[pe] != NULL );

      for (int bank=0; bank<NUM_RAM_BANKS; bank++)
      {

        ram[pe][bank] = new Int16[ramBankSize]; assert( ram[pe][bank] != NULL);

      } // for

      ramEnStack[pe] = new bool[hwStackSize]; assert( ramEnStack[pe] != NULL );

    } // for

    broadcastBusValue.setValue( 0 );
    leftBusValue = rightBusValue.setValue( 0 );
    leftInjectorsEnabled = rightInjectorsEnabled = false;

    toggleStackEn = pushStackEn = popStackEn = false;
    stackPtr = 0;
    nextAvailableAddressA = nextAvailableAddressB = 0;

    disableAdder();
    disableByteSelect();
    setShifter( noShift );

    cycleCount = 0;

    for ( p=0; p < numberOfPEs; p++ ) {
        leftShiftBusValue[ p ].setValue( 0 );
        rightShiftBusValue[ p ].setValue( 0 );
        acc2[p]     = acc1[p]     = acc0[p].setValue( 0 );
        acc2Next[p] = acc1Next[p] = acc0Next[p].setValue( 0 );
        shReg[p] = shRegNext[p].setValue( 0 );
        broadBusReg[p] = broadBusRegNext[p].setValue( 0 );
        multInput1[p] = multInput1Next[p].setValue( 0 );
        multInput2[p] = multInput2Next[p].setValue( 0 );
        multMidReg[p] = multMidRegNext[p].setValue( 0 );
        multOutput[p] = multOutputNext[p].setValue( 0 );
        for ( j=0; j < ramBankSize; j++ ) {
            ram[p][BANK_A][j] = ram[p][BANK_B][j].setValue( 0 );
        }
        for ( j=0; j<hwStackSize; j++ ) {
            ramEnStack[p][j] = false;
        }
        ramEnStack[p][stackPtr] = true;
    }
}

DSPRAMClass::~DSPRAMClass()
{

  delete[] acc0;
  delete[] acc0Next;
  delete[] acc1;
  delete[] acc1Next;
  delete[] acc2;
  delete[] acc2Next;

  delete[] multInput1;
  delete[] multInput1Next;
  delete[] multInput2;
  delete[] multInput2Next;
  delete[] multMidReg;
  delete[] multMidRegNext;
  delete[] multOutput;
```

```
    delete[] multOutputNext;

    delete[] broadBusReg;
    delete[] broadBusRegNext;
    delete[] leftShiftBusValue;
    delete[] rightShiftBusValue;

    delete[] ramEnNext;

    for (int pe=0; pe<numberOfPEs; pe++)
    {

      for (int bank=0; bank<NUM_RAM_BANKs; bank++)
      {

        delete[] ram[pe][bank];

      } // for

      delete[] ram[pe];
      delete[] ramEnStack[pe];

    } // for

    delete[] shReg;
    delete[] shRegNext;
    delete[] ram;
    delete[] ramEnStack;

} // DSPRAMClass Destructor

// Memory allocation functions.  Return value is the base address.

int DSPRAMClass::intAlloc( int bank_num )
// Used to allocated one 16-bit parallel variable
{
    if ( bank_num == BANK_A )
        return nextAvailableAddressA++;
    else
        return nextAvailableAddressB++;
}

int DSPRAMClass::intVectorAlloc( int bank_num, int vectorLength )
{
    int addr;

    assert( vectorLength > 0 );

    if ( bank_num == BANK_A ) {
        addr = nextAvailableAddressA;
        nextAvailableAddressA += vectorLength;
        assert( nextAvailableAddressA < ramBankSize );
    } else {
        addr = nextAvailableAddressB;
        nextAvailableAddressB += vectorLength;
        assert( nextAvailableAddressB < ramBankSize );
    } // end if

    return addr;
}


// Microprocessor port write and read operations

void DSPRAMClass::write( Int16 data, int pe_num, int bank_num, int address )
{
    ram[pe_num][bank_num][address] = data;
}

Int16 DSPRAMClass::read( int pe_num, int bank_num, int address )
{
    return ram[pe_num][bank_num][address];
}

// Shift bus operations

#ifdef SHIFTBUS_16

void DSPRAMClass::injectIntoLeft( Int16 injectedValue )
{
    leftInjectorsEnabled  = true;
    rightInjectorsEnabled = false;
    leftBusValue = injectedValue;
}

void DSPRAMClass::injectIntoRight( Int16 injectedValue )
{
    leftInjectorsEnabled  = false;
    rightInjectorsEnabled = true;
    rightBusValue = injectedValue;
}
```

```
Int16 DSPRAMClass::extractFromLeft()
{
    return leftBusValue;
}

Int16 DSPRAMClass::extractFromRight()
{
    return rightBusValue;
}

void DSPRAMClass::rightBusFromAcc( WordPosition position )
{

  // Set the initial values
  leftShiftBusValue[0] = leftBusValue;

  switch (position)
  {

    case WORD_0 : rightShiftBusValue[0] = acc0[0]; break;
    case WORD_1 : rightShiftBusValue[0] = acc1[0]; break;
    case WORD_2 : rightShiftBusValue[0] = acc2[0]; break;

    default :

      cerr << "Illegal_Word_position" << endl;
      abort();

    break;

  } // switch

  if ( numberOfPEs > 1 )
  {

    for ( int p=1; p < numberOfPEs; p++ )
    {

      leftShiftBusValue[p] = rightShiftBusValue[p-1];

      switch (position)
      {

        case WORD_0 : rightShiftBusValue[p] = acc0[p]; break;
        case WORD_1 : rightShiftBusValue[p] = acc1[p]; break;
        case WORD_2 : rightShiftBusValue[p] = acc2[p]; break;

        default :

          cerr << "Illegal_Word_position" << endl;
          abort();

        break;

      } // switch

    } // for

  } // if

  rightBusValue = rightShiftBusValue[ numberOfPEs - 1 ];

} // rightBusFromAcc

void DSPRAMClass::leftBusFromAcc( WordPosition position )
{

  // Set-up initial values
  rightShiftBusValue[numberOfPEs - 1] = rightBusValue;

  for ( int p=numberOfPEs-1; p >= 0; p--)
  {

    switch (position)
    {

      case WORD_0 : leftShiftBusValue[p] = acc0[p]; break;
      case WORD_1 : leftShiftBusValue[p] = acc1[p]; break;
      case WORD_2 : leftShiftBusValue[p] = acc2[p]; break;

      default :

        cerr << "Invalid_Word_Position_specified" << endl;
        abort();

      break;

    } // switch

    if ( p == 0 )
    {
```

```
      leftBusValue = leftShiftBusValue[0];

    } // if
    else
    {

      rightShiftBusValue[p-1] = leftShiftBusValue[p];

    } // else

  } // for

} // leftBusFromAcc

#ifdef ENHANCED_ROUTING

void DSPRAMClass::rightBusFromInputA( void )
{

  // Set the initial values
  leftShiftBusValue[0]  = leftBusValue;
  rightShiftBusValue[0] = multInput1[0];

  for ( int p=1; p < numberOfPEs; p++ )
  {

    leftShiftBusValue[p] = rightShiftBusValue[p-1];
    rightShiftBusValue[p] = multInput1[p];

  } // for

  rightBusValue = rightShiftBusValue[ numberOfPEs - 1 ];

} // rightBusFromInputA

void DSPRAMClass::leftBusFromInputA( void )
{

  // Set-up initial values
  rightShiftBusValue[numberOfPEs - 1] = rightBusValue;

  for ( int p=numberOfPEs-1; p >= 0; p-- )
  {

    leftShiftBusValue[p] = multInput1[p];

    if ( p == 0 )
    {

      leftBusValue = leftShiftBusValue[0];

    } // if
    else
    {

      rightShiftBusValue[p-1] = leftShiftBusValue[p];

    } // else

  } // for

} // leftBusFromInputA

void DSPRAMClass::saveInputA( int bankNumber, int address )
{

  int p;

  for ( p=0; p<numberOfPEs; p++ )
  {

    if ( byteSelectEnabled )
    {

      int temp = ram[p][bankNumber][address];
      temp = (temp >> useByte) & 0x00FF;
      multInput1Next[p] = temp;

    }
    else
    {

      multInput1Next[p] = ram[p][bankNumber][address];

    }

  } // for

} // saveInputA

#endif
```

```
#else  // 48-bit shift bus

void DSPRAMClass::injectIntoLeft( Int48 injectedValue )
{
    leftInjectorsEnabled  = true;
    rightInjectorsEnabled = false;
    leftBusValue = injectedValue;
}

void DSPRAMClass::injectIntoRight( Int48 injectedValue )
{
    leftInjectorsEnabled  = false;
    rightInjectorsEnabled = true;
    rightBusValue = injectedValue;
}

Int48 DSPRAMClass::extractFromLeft()
{
    return leftBusValue;
}

Int48 DSPRAMClass::extractFromRight()
{
    return rightBusValue;
}

void DSPRAMClass::rightBusFromAcc()
{

    leftShiftBusValue[0]  = leftBusValue;
    rightShiftBusValue[0].setValue( acc2[0], acc1[0], acc0[0] );

    if ( numberOfPEs > 1 ) {
        for ( int p=1; p < numberOfPEs; p++ ) {
            leftShiftBusValue[p].setValue( acc2[p-1], acc1[p-1], acc0[p-1] );
            rightShiftBusValue[p].setValue( acc2[p], acc1[p], acc0[p] );
        }
    }

    rightBusValue = rightShiftBusValue[ numberOfPEs - 1 ];
}

void DSPRAMClass::leftBusFromAcc()
{

  rightShiftBusValue[ numberOfPEs - 1 ] = rightBusValue;
  leftShiftBusValue[numberOfPEs-1].setValue( acc2[numberOfPEs-1], acc1[numberOfPEs-1], acc0[numberOfPEs-1] );

  for ( int p=(numberOfPEs - 1); p >= 0; p--)
  {

    leftShiftBusValue[p].setValue( acc2[p], acc1[p], acc0[p] );

    if ( p==0 )
    {

      leftBusValue = leftShiftBusValue[ 0 ];

    } // if
    else
    {

      rightShiftBusValue[p-1] = leftShiftBusValue[p];

    } // else

  } // for

} // DSPRAMClass::leftBusFromAcc ()

#endif

void DSPRAMClass::rightBusFromShReg( void )
{

  // Set-up initial values
  leftShiftBusValue[0]  = leftBusValue;
  rightShiftBusValue[0] = shReg[0];

  if ( numberOfPEs > 1 )
  {

    for ( int p=1; p < numberOfPEs; p++ )
    {

      leftShiftBusValue[p]  = rightShiftBusValue[p-1];
      rightShiftBusValue[p] = shReg[p];

    } // for

  } // if
```

```
  rightBusValue = rightShiftBusValue[ numberOfPEs - 1 ];

} // rightBusFromShReg

void DSPRAMClass::leftBusFromShReg( void )
{
  // Set-up initial conditions
  rightShiftBusValue[ numberOfPEs - 1 ] = rightBusValue;
  leftShiftBusValue [ numberOfPEs - 1 ]  = shReg[ numberOfPEs - 1 ];

  if ( numberOfPEs > 1 )
  {

    for ( int p=0; p < numberOfPEs - 1; p++ )
    {

      leftShiftBusValue[p]  = shReg[p];
      rightShiftBusValue[p] = leftShiftBusValue[p+1];

    } // for

  } // if

  leftBusValue = leftShiftBusValue[ 0 ];

} // leftBusFromShReg

// Broadcast bus operations

void DSPRAMClass::injectOntoBroadcastBus ( Int16 injectedValue )
{
    broadcastBusValue = injectedValue;
}

Int16 DSPRAMClass::extractFromBroadcastBus()
{
    return broadcastBusValue;
}

// Register operations

void DSPRAMClass::regLoadBroadcastBus()
{
    for ( int p=0; p < numberOfPEs; p++ ) {
        broadBusRegNext[p] = broadcastBusValue;
    }
}

void DSPRAMClass::regWriteBroadcastBus()
{
    broadcastBusValue.setValue( 65535 );

    for ( int p=0; p < numberOfPEs; p++ ) {
        if (ramEnStack[p][stackPtr]) //only output something if ram is enabled otherwise 'Z'
                broadcastBusValue = broadcastBusValue & broadBusReg[p];
    }
}

void DSPRAMClass::regLoadDataA( int addressA )
{
    for ( int p=0; p < numberOfPEs; p++ ) {
        broadBusRegNext[p] = ram[p][BANK_A][addressA];
    }
}

void DSPRAMClass::regLoadDataB( int addressB )
{
    for ( int p=0; p < numberOfPEs; p++ ) {
        broadBusRegNext[p] = ram[p][BANK_B][addressB];
    }
}

void DSPRAMClass::regLoadAccWord( int accByteNumber )
// argument is either 2, 1, or 0
{
    for ( int p=0; p < numberOfPEs; p++ ) {
        switch ( accByteNumber ) {
            case 2: broadBusRegNext[p] = acc2[p];
                    break;
            case 1: broadBusRegNext[p] = acc1[p];
                    break;
            case 0: broadBusRegNext[p] = acc0[p];
                    break;
        }
    }
}

// Accumulator operations
void DSPRAMClass::accClear()
// all three accumulator words cleared
{
```

```
    for ( int p=0; p < numberOfPEs; p++ ) {
        acc2Next[p] = acc1Next[p] = acc0Next[p].setValue( 0 );
    }
}

#ifdef SHIFTBUS_16

void DSPRAMClass::accLoadFromLeft( WordPosition position )
{

  for ( int p=0; p < numberOfPEs; p++ )
  {

    switch (position)
    {

      case WORD_0 : acc0Next[p] = leftShiftBusValue[p]; break;
      case WORD_1 : acc1Next[p] = leftShiftBusValue[p]; break;
      case WORD_2 : acc2Next[p] = leftShiftBusValue[p]; break;

      default :

        cerr << "Invalid_word_position_specified" << endl;
        abort();

      break;

    } // switch

  } // for

} // accLoadFromLeft

void DSPRAMClass::accLoadFromRight( WordPosition position )
{

  for ( int p=0; p < numberOfPEs; p++ )
  {

    switch (position)
    {

      case WORD_0 : acc0Next[p] = rightShiftBusValue[p]; break;
      case WORD_1 : acc1Next[p] = rightShiftBusValue[p]; break;
      case WORD_2 : acc2Next[p] = rightShiftBusValue[p]; break;

      default :

        cerr << "Invalid_word_position_specified" << endl;
        abort();

      break;

    } // switch

  } // for

} // for

#else

void DSPRAMClass::accLoadFromLeft()
// all three accumulator words loaded
{
    for ( int p=0; p < numberOfPEs; p++ ) {
        leftShiftBusValue[p].partition( acc2Next[p], acc1Next[p], acc0Next[p] );
    }
}

void DSPRAMClass::accLoadFromRight()
// all three accumulator words loaded
{
    for ( int p=0; p < numberOfPEs; p++ ) {
        rightShiftBusValue[p].partition( acc2Next[p], acc1Next[p], acc0Next[p] );
    }
}

#endif

void DSPRAMClass::accLoadShifter()
// all three accumulator words loaded
{
    Int48 shiftedProduct, accWord;

    for ( int p=0; p < numberOfPEs; p++ ) {
        shiftedProduct = multOutput[p];          // convert from Int32 to Int48

        if ( adderEnabled ) {
            accWord.setValue( acc2[p], acc1[p], acc0[p] );
            shiftedProduct = shiftedProduct + accWord;
        }
```

```
        shiftedProduct.shift( currentShiftValue );
        shiftedProduct.partition( acc2Next[p], acc1Next[p], acc0Next[p] );
    }
}

void DSPRAMClass::acc0LoadShifter()
{
    Int16 dummy1, dummy2;
    Int48 shiftedProduct, accWord;

    for ( int p=0; p < numberOfPEs; p++ ) {
        shiftedProduct = multOutput[p];
        if ( adderEnabled ) {
            accWord.setValue( acc2[p], acc1[p], acc0[p] );
            shiftedProduct = shiftedProduct + accWord;
        }
        shiftedProduct.shift( currentShiftValue );
        shiftedProduct.partition( dummy1, dummy2, acc0Next[p] );
    }
}

void DSPRAMClass::acc1LoadShifter()
{
    Int16 dummy1, dummy2;
    Int48 shiftedProduct, accWord;

    for ( int p=0; p < numberOfPEs; p++ ) {
        shiftedProduct = multOutput[p];
        if ( adderEnabled ) {
            accWord.setValue( acc2[p], acc1[p], acc0[p] );
            shiftedProduct = shiftedProduct + accWord;
        }
        shiftedProduct.shift( currentShiftValue );
        shiftedProduct.partition( dummy1, acc1Next[p], dummy2 );
    }
}

void DSPRAMClass::acc2LoadShifter()
{
    Int16 dummy1, dummy2;
    Int48 shiftedProduct, accWord;

    for ( int p=0; p < numberOfPEs; p++ ) {
        shiftedProduct = multOutput[p];
        if ( adderEnabled ) {
            accWord.setValue( acc2[p], acc1[p], acc0[p] );
            shiftedProduct = shiftedProduct + accWord;
        }
        shiftedProduct.shift( currentShiftValue );
        shiftedProduct.partition( acc2Next[p], dummy1, dummy2 );
    }
}

// Shift register operations
void DSPRAMClass::shRegClear()
// all three accumulator words cleared
{

  for ( int p=0; p < numberOfPEs; p++ )
  {

#ifdef SHIFTBUS_16

    shRegNext[p] = Int16(0);

#else

    shRegNext[p] = Int48(0);

#endif

  } // for

} // shRegClear

void DSPRAMClass::shRegLoadFromLeft( void )
{

  for ( int p=0; p < numberOfPEs; p++ )
  {

    shRegNext[p] = leftShiftBusValue[p];

  } // for

} // shRegLoadFromLeft

void DSPRAMClass::shRegLoadFromRight( void )
{

  for ( int p=0; p < numberOfPEs; p++ )
```

```
  {

    shRegNext[p] = rightShiftBusValue[p];

  } // for

} // shRegLoadFromRight

#ifdef SHIFTBUS_16

void DSPRAMClass::shRegLoadShifter( WordPosition position )
// all three accumulator words loaded
{

  Int48 accWord;
  Int48 sr;
  Int16 words[NUMBER_OF_WORD_POSITIONS];

  for ( int p=0; p < numberOfPEs; p++ )
  {

    sr = multOutput[p];

    if ( adderEnabled )
    {

      accWord.setValue( acc2[p], acc1[p], acc0[p] );
      sr = sr + accWord;

    } // if

    sr.shift( currentShiftValue );

    sr.partition( words[WORD_2], words[WORD_1], words[WORD_0] );

    shRegNext[p] = words[position];

  } // for

} // shRegLoadShifter

#else

void DSPRAMClass::shRegLoadShifter()
// all three accumulator words loaded
{

  Int48 accWord;

  for ( int p=0; p < numberOfPEs; p++ )
  {

    shRegNext[p] = multOutput[p];

    if ( adderEnabled )
    {

      accWord.setValue( acc2[p], acc1[p], acc0[p] );
      shRegNext[p] = shRegNext[p] + accWord;

    } // if

    shRegNext[p].shift( currentShiftValue );

  } // for

} // shRegLoadShifter

#endif

void DSPRAMClass::multAxB( int addressA, int addressB )
{
    for ( int p=0; p < numberOfPEs; p++ ) {

        if (ramEnStack[p][stackPtr] == true)
        {
            if ( byteSelectEnabled )
            {

              int temp = ram[p][BANK_A][addressA];
              temp = (temp >> useByte) & 0x00FF;
              multInput1Next[p] = temp;

            }
            else
            {

              multInput1Next[p] = ram[p][BANK_A][addressA];

            }

            multInput2Next[p] = ram[p][BANK_B][addressB];
```

```
        }
        else
        {
            multInput1Next[p] = 0;
            multInput2Next[p] = 0;
        }
    }
}

void DSPRAMClass::multBxB( int addressB )
{
    for ( int p=0; p < numberOfPEs; p++ ) {
        if (ramEnStack[p][stackPtr] == true)
        {
                multInput1Next[p] = ram[p][BANK_B][addressB];
                multInput2Next[p] = ram[p][BANK_B][addressB];
        }
        else
        {
                multInput1Next[p] = 0;
                multInput2Next[p] = 0;
        }
    }
}

void DSPRAMClass::multAxR( int addressA )
{
    for ( int p=0; p < numberOfPEs; p++ )
    {
        if (ramEnStack[p][stackPtr] == true)
        {

            if ( byteSelectEnabled )
            {

              int temp = ram[p][BANK_A][addressA];
              temp = (temp >> useByte) & 0x00FF;
              multInput1Next[p] = temp;

            }
            else
            {

              multInput1Next[p] = ram[p][BANK_A][addressA];

            }

        }
        else
        {
                multInput1Next[p] = 0;
        }
        multInput2Next[p] = broadBusReg[p];
    }
}

void DSPRAMClass::multBxR( int addressB )
{
    for ( int p=0; p < numberOfPEs; p++ ) {
        if (ramEnStack[p][stackPtr] == true)
        {
                multInput1Next[p] = ram[p][BANK_B][addressB];
        }
        else
        {
                multInput1Next[p] = 0;
        }
        multInput2Next[p] = broadBusReg[p];
    }
}

// Adder control functions  (new to Version 2.1)

void DSPRAMClass::enableAdder()
{
    adderEnabled = true;
}

void DSPRAMClass::disableAdder()
{
    adderEnabled = false;
}

// Shifter control functions  (new to Version 2.1)

void DSPRAMClass::setShifter( ShiftValue sv )
{
    currentShiftValue = sv;
}

// Memory load operations
```

```
void DSPRAMClass::memLoadAccWord( int bank_num, int address, int accByteNumber )
{
    for ( int p=0; p < numberOfPEs; p++ ) {

      if (ramEnStack[p][stackPtr])
      {
              switch ( accByteNumber ) {
              case ACC_0:
                  ram[p][bank_num][address] = acc0[p];
                  break;
              case ACC_1:
                  ram[p][bank_num][address] = acc1[p];
                  break;
              case ACC_2:
                  ram[p][bank_num][address] = acc2[p];
                  break;
              }
      }
    }

}

#ifdef ENHANCED_ROUTING

void DSPRAMClass::memLoadInputA( int bank_num, int address )
{
  for ( int p=0; p < numberOfPEs; p++ )
  {

    if (ramEnStack[p][stackPtr])
    {

      ram[p][bank_num][address] = multInput1[p];

    } // if

  } // for

} // memLoadInputA

#endif

// Memory enable operations.  The bank_num argument is either 0 or 1.

void DSPRAMClass::pushEnableAcqB( int addressA, int addressB )
{
    pushStackEn = true;

    for ( int p=0; p < numberOfPEs; p++ ) {

        if (ramEnStack[p][stackPtr])
        {
            if ( byteSelectEnabled )
            {
                Int16 temp = ram[p][BANK_A][addressA];
                temp = (temp >> useByte) & 0x00FF;

                if ( temp == ram[p][BANK_B][addressB] )
                {

                    ramEnNext[p] = true;

                } // if
                else
                {

                    ramEnNext[p] = false;

                } // else

            } // if
            else
            {

                if ( ram[p][BANK_A][addressA] == ram[p][BANK_B][addressB] )
                {

                    ramEnNext[p] = true;

                } // if
                else
                {

                    ramEnNext[p] = false;

                } // else

            } // else
```

```
        } // if
        else
        {

            ramEnNext[p] = false;

        } // else

    } // for

}

void DSPRAMClass::pushEnableAeqR( int addressA )
{
    pushStackEn = true;

    for ( int p=0; p < numberOfPEs; p++ )
    {

        if (ramEnStack[p][stackPtr])
        {

            if ( byteSelectEnabled )
            {

                Int16 temp = ram[p][BANK_A][addressA];
                temp = (temp >> useByte) & 0x00FF;

                if ( temp == broadBusReg[p] )
                {

                    ramEnNext[p] = true;

                } // if
                else
                {

                    ramEnNext[p] = false;

                } // else

            } // if
            else
            {

                if ( ram[p][BANK_A][addressA] == broadBusReg[p] )
                {

                    ramEnNext[p] = true;

                } // if
                else
                {

                    ramEnNext[p] = false;

                } // else

            } // else

        } // if
        else
        {

            ramEnNext[p] = false;

        } // else

    } // for

}

void DSPRAMClass::pushEnableBeqR( int addressB )
{
    pushStackEn = true;

    for ( int p=0; p < numberOfPEs; p++ ) {
        if (ramEnStack[p][stackPtr])
                if ( ram[p][BANK_B][addressB] == broadBusReg[p] )
                    ramEnNext[p] = true;
                else
                    ramEnNext[p] = false;
        else
                ramEnNext[p] = false;
    }
}

void DSPRAMClass::pushEnableAltB( int addressA, int addressB )
{
    pushStackEn = true;

    for ( int p=0; p < numberOfPEs; p++ ) {
```

```
        if (ramEnStack[p][stackPtr])
        {
            if ( byteSelectEnabled )
            {
                Int16 temp = ram[p][BANK_A][addressA];
                temp = (temp >> useByte) & 0x00FF;
                if ( temp < ram[p][BANK_B][addressB] )
                {
                    ramEnNext[p] = true;
                } // if
                else
                {
                    ramEnNext[p] = false;
                } // else
            } // if
            else
            {
                if ( ram[p][BANK_A][addressA] < ram[p][BANK_B][addressB] )
                {
                    ramEnNext[p] = true;
                } // if
                else
                {
                    ramEnNext[p] = false;
                } // else
            } // else
        } // if
        else
        {
            ramEnNext[p] = false;
        } // else
    } // for
}
void DSPRAMClass::pushEnableAltR( int addressA )
{
    pushStackEn = true;
    for ( int p=0; p < numberOfPEs; p++ ) {
        if (ramEnStack[p][stackPtr])
        {
            if ( byteSelectEnabled )
            {
                Int16 temp = ram[p][BANK_A][addressA];
                temp = (temp >> useByte) & 0x00FF;
                if ( temp < broadBusReg[p] )
                {
                    ramEnNext[p] = true;
                } // if
                else
                {
                    ramEnNext[p] = false;
                } // else
            } // if
            else
            {
                if ( ram[p][BANK_A][addressA] < broadBusReg[p] )
                {
                    ramEnNext[p] = true;
                } // if
```

```
                else
                {

                    ramEnNext[p] = false;

                } // else

            } // else

        } // if
        else
        {

            ramEnNext[p] = false;

        } // else

    } // for

}

void DSPRAMClass::pushEnableBltR( int addressB )
{
    pushStackEn = true;

    for ( int p=0; p < numberOfPEs; p++ ) {
        if (ramEnStack[p][stackPtr])
                if ( ram[p][BANK_B][addressB] < broadBusReg[p] )
                    ramEnNext[p] = true;
                else
                    ramEnNext[p] = false;
        else
                ramEnNext[p] = false;
    }
}

void DSPRAMClass::pushEnableAgtB( int addressA, int addressB )
{
    pushStackEn = true;

    for ( int p=0; p < numberOfPEs; p++ ) {

        if (ramEnStack[p][stackPtr])
        {

            if ( byteSelectEnabled )
            {

                Int16 temp = ram[p][BANK_A][addressA];
                temp = (temp >> useByte) & 0x00FF;

                if ( temp > ram[p][BANK_B][addressB] )
                {

                    ramEnNext[p] = true;

                } // if
                else
                {

                    ramEnNext[p] = false;

                } // else

            } // if
            else
            {

                if ( ram[p][BANK_A][addressA] > ram[p][BANK_B][addressB] )
                {

                    ramEnNext[p] = true;

                } // if
                else
                {

                    ramEnNext[p] = false;

                } // else

            } // else

        } // if
        else
        {

            ramEnNext[p] = false;

        } // else

    } // for
```

```
}

void DSPRAMClass::pushEnableAgeB( int addressA, int addressB )
{
    pushStackEn = true;

    for ( int p=0; p < numberOfPEs; p++ ) {

        if (ramEnStack[p][stackPtr])
        {

            if ( byteSelectEnabled )
            {

                Int16 temp = ram[p][BANK_A][addressA];
                temp = (temp >> useByte) & 0x00FF;

                if ( temp >= ram[p][BANK_B][addressB] )
                {

                    ramEnNext[p] = true;

                } // if
                else
                {

                    ramEnNext[p] = false;

                } // else

            } // if
            else
            {

                if ( ram[p][BANK_A][addressA] >= ram[p][BANK_B][addressB] )
                {

                    ramEnNext[p] = true;

                } // if
                else
                {

                    ramEnNext[p] = false;

                } // else

            } // else

        } // if
        else
        {

            ramEnNext[p] = false;

        } // else

    } // for

}

void DSPRAMClass::pushEnableAgtR( int addressA )
{
    pushStackEn = true;

    for ( int p=0; p < numberOfPEs; p++ ) {

        if (ramEnStack[p][stackPtr])
        {

            if ( byteSelectEnabled )
            {

                Int16 temp = ram[p][BANK_A][addressA];
                temp = (temp >> useByte) & 0x00FF;

                if ( temp > broadBusReg[p] )
                {

                    ramEnNext[p] = true;

                } // if
                else
                {

                    ramEnNext[p] = false;

                } // else

            } // if
            else
```

```
            {
                if ( ram[p][BANK_A][addressA] > broadBusReg[p] )
                {
                    ramEnNext[p] = true;
                } // if
                else
                {
                    ramEnNext[p] = false;
                } // else
            } // else
        } // if
        else
        {
            ramEnNext[p] = false;
        } // else
    } // for
}
void DSPRAMClass::pushEnableAgeR( int addressA )
{
    pushStackEn = true;
    for ( int p=0; p < numberOfPEs; p++ ) {
        if (ramEnStack[p][stackPtr])
        {
            if ( byteSelectEnabled )
            {
                Int16 temp = ram[p][BANK_A][addressA];
                temp = (temp >> useByte) & 0x00FF;
                if ( temp >= broadBusReg[p] )
                {
                    ramEnNext[p] = true;
                } // if
                else
                {
                    ramEnNext[p] = false;
                } // else
            } // if
            else
            {
                if ( ram[p][BANK_A][addressA] >= broadBusReg[p] )
                {
                    ramEnNext[p] = true;
                } // if
                else
                {
                    ramEnNext[p] = false;
                } // else
            } // else
        } // if
        else
        {
            ramEnNext[p] = false;
        } // else
    } // for
}
void DSPRAMClass::pushEnableBgtR ( int addressB )
{
    pushStackEn = true;
```

```
    for ( int p=0; p < numberOfPEs; p++ ) {
        if (ramEnStack[p][stackPtr])
                if ( ram[p][BANK_B][addressB] > broadBusReg[p] )
                    ramEnNext[p] = true;
                else
                    ramEnNext[p] = false;
        else
                ramEnNext[p] = false;
    }
}

void DSPRAMClass::toggleEnable()
{

  // Top of stack should only be toggled if the previous one
  // was enabled.  Plus, should only happen on clock edge...
  toggleStackEn = true;

} // toggleEnable

void DSPRAMClass::popEnable()
{
    popStackEn = true;
}

void DSPRAMClass::pushEnableBusResult()
{
    int p;
    Int16 minValue( MAX_INT16 );

    pushStackEn = true;

    for ( p=0; p<numberOfPEs; p++ ) {
        if (ramEnStack[p][stackPtr])
                if ( broadBusReg[p] < minValue )
                    minValue = broadBusReg[p];
        else // won't output anything except high impedence when Ram is disable
                ramEnNext[p] = false;
    }

    for ( p=0; p<numberOfPEs; p++ ) {
        if (ramEnStack[p][stackPtr])
                if ( minValue == broadBusReg[p] )
                    ramEnNext[p] = true;
                else
                    ramEnNext[p] = false;
    }
    cout << "Comparing broadcast bus for result value" << "---" << minValue<<endl;
}

void DSPRAMClass::printRegisters()
{
    cout << endl << "Clock cycle #" << getCycleCount() << endl;
    cout << "N=" << numberOfPEs << endl;

    cout << "      ramA[0]=" << setw(13) << setfill('_') << ram[0][BANK_A][0]
         << "      ramA[1]=" << setw(13) << setfill('_') << ram[1][BANK_A][0]
         << "...."
         << "    ramA[N-2]=" << setw(13) << setfill('_') << ram[numberOfPEs - 2][BANK_A][0]
         << "    ramA[N-1]=" << setw(13) << setfill('_') << ram[numberOfPEs - 1][BANK_A][0] << endl;
    cout << "                " << setw(13) << setfill('_') << ram[0][BANK_A][1]
         << "                " << setw(13) << setfill('_') << ram[1][BANK_A][1]
         << "...."
         << "                " << setw(13) << setfill('_') << ram[numberOfPEs - 2][BANK_A][1]
         << "                " << setw(13) << setfill('_') << ram[numberOfPEs - 1][BANK_A][1] << endl;
    cout << "                " << setw(13) << setfill('_') << ram[0][BANK_A][2]
         << "                " << setw(13) << setfill('_') << ram[1][BANK_A][2]
         << "...."
         << "                " << setw(13) << setfill('_') << ram[numberOfPEs - 2][BANK_A][2]
         << "                " << setw(13) << setfill('_') << ram[numberOfPEs - 1][BANK_A][2] << endl;
    cout << "      ramB[0]=" << setw(13) << setfill('_') << ram[0][BANK_B][0]
         << "      ramB[1]=" << setw(13) << setfill('_') << ram[1][BANK_B][0]
         << "...."
         << "    ramB[N-2]=" << setw(13) << setfill('_') << ram[numberOfPEs - 2][BANK_B][0]
         << "    ramB[N-1]=" << setw(13) << setfill('_') << ram[numberOfPEs - 1][BANK_B][0] << endl;
    cout << "                " << setw(13) << setfill('_') << ram[0][BANK_B][1]
         << "                " << setw(13) << setfill('_') << ram[1][BANK_B][1]
         << "...."
         << "                " << setw(13) << setfill('_') << ram[numberOfPEs - 2][BANK_B][1]
         << "                " << setw(13) << setfill('_') << ram[numberOfPEs - 1][BANK_B][1] << endl;
    cout << "                " << setw(13) << setfill('_') << ram[0][BANK_B][2]
         << "                " << setw(13) << setfill('_') << ram[1][BANK_B][2]
         << "...."
         << "                " << setw(13) << setfill('_') << ram[numberOfPEs - 2][BANK_B][2]
         << "                " << setw(13) << setfill('_') << ram[numberOfPEs - 1][BANK_B][2] << endl;
    cout << "    Enable[0]=" << setw(13) << setfill('_') << ramEnStack[0][stackPtr]
         << "    Enable[1]=" << setw(13) << setfill('_') << ramEnStack[1][stackPtr]
         << "...."
         << "  Enable[N-2]=" << setw(13) << setfill('_') << ramEnStack[numberOfPEs - 2][stackPtr]
         << "  Enable[N-1]=" << setw(13) << setfill('_') << ramEnStack[numberOfPEs - 1][stackPtr] << endl;
    cout << "      bbRg[0]=" << setw(13) << setfill('_') << broadBusReg[0]
```

```
        << "______bbRg[1]_=_" << setw(13) << setfill('_') << broadBusReg[1]
        << "_...."
        << "____bbRg[N-2]_=_" << setw(13) << setfill('_') << broadBusReg[numberOfPEs - 2]
        << "____bbRg[N-1]_=_" << setw(13) << setfill('_') << broadBusReg[numberOfPEs - 1] << endl;
    cout << "______shRg[0]_=_" << setw(13) << setfill('_') << shReg[0]
        << "______shRg[1]_=_" << setw(13) << setfill('_') << shReg[1]
        << "_...."
        << "____shRg[N-2]_=_" << setw(13) << setfill('_') << shReg[numberOfPEs - 2]
        << "____shRg[N-1]_=_" << setw(13) << setfill('_') << shReg[numberOfPEs - 1] << endl;
    cout << "___multIn1[0]_=_" << setw(13) << setfill('_') << multInput1[0]
        << "___multIn1[1]_=_" << setw(13) << setfill('_') << multInput1[1]
        << "_...."
        << "_multIn1[N-2]_=_" << setw(13) << setfill('_') << multInput1[numberOfPEs - 2]
        << "_multIn1[N-1]_=_" << setw(13) << setfill('_') << multInput1[numberOfPEs - 1] << endl;
    cout << "___multIn2[0]_=_" << setw(13) << setfill('_') << multInput2[0]
        << "___multIn2[1]_=_" << setw(13) << setfill('_') << multInput2[1]
        << "_...."
        << "_multIn2[N-1]_=_" << setw(13) << setfill('_') << multInput2[numberOfPEs - 2]
        << "_multIn2[N-2]_=_" << setw(13) << setfill('_') << multInput2[numberOfPEs - 1] << endl;
    cout << "___multMid[0]_=_" << setw(13) << setfill('_') << multMidReg[0]
        << "___multMid[1]_=_" << setw(13) << setfill('_') << multMidReg[1]
        << "_...."
        << "_multMid[N-1]_=_" << setw(13) << setfill('_') << multMidReg[numberOfPEs - 2]
        << "_multMid[N-2]_=_" << setw(13) << setfill('_') << multMidReg[numberOfPEs - 1] << endl;
    cout << "___multOut[0]_=_" << setw(13) << setfill('_') << multOutput[0]
        << "___multOut[1]_=_" << setw(13) << setfill('_') << multOutput[1]
        << "_...."
        << "_multOut[N-1]_=_" << setw(13) << setfill('_') << multOutput[numberOfPEs - 2]
        << "_multOut[N-2]_=_" << setw(13) << setfill('_') << multOutput[numberOfPEs - 1] << endl;
    cout << "_______acc[0]_=_" << setw(13) << setfill('_') << Int48( acc2[0], acc1[0], acc0[0] )
        << "_______acc[1]_=_" << setw(13) << setfill('_') << Int48( acc2[1], acc1[1], acc0[1] )
        << "_...."
        << "_____acc[N-1]_=_" << setw(13) << setfill('_') << Int48( acc2[numberOfPEs - 2], acc1[numberOfPEs - 2], acc0[numberOfPEs - 2] )
        << "_____acc[N-2]_=_" << setw(13) << setfill('_') << Int48( acc2[numberOfPEs - 1], acc1[numberOfPEs - 1], acc0[numberOfPEs - 1] ) <<

}

void DSPRAMClass::dumpAcc()
{

  assert( numberOfPEs >= 64 );

  cout << "PE_#0_ACC_=_" << Int48( acc2[0], acc1[0], acc0[0] ) << endl;
  cout << "PE_#1_ACC_=_" << Int48( acc2[1], acc1[1], acc0[1] ) << endl;
  cout << "PE_#2_ACC_=_" << Int48( acc2[2], acc1[2], acc0[2] ) << endl;
  cout << "PE_#3_ACC_=_" << Int48( acc2[3], acc1[3], acc0[3] ) << endl;
  cout << "PE_#4_ACC_=_" << Int48( acc2[4], acc1[4], acc0[4] ) << endl;
  cout << "PE_#5_ACC_=_" << Int48( acc2[5], acc1[5], acc0[5] ) << endl;
  cout << "PE_#6_ACC_=_" << Int48( acc2[6], acc1[6], acc0[6] ) << endl;
  cout << "PE_#7_ACC_=_" << Int48( acc2[7], acc1[7], acc0[7] ) << endl;
  cout << "PE_#8_ACC_=_" << Int48( acc2[8], acc1[8], acc0[8] ) << endl;
  cout << "PE_#9_ACC_=_" << Int48( acc2[9], acc1[9], acc0[9] ) << endl;
  cout << "PE_#10_ACC_=_" << Int48( acc2[10], acc1[10], acc0[10] ) << endl;
  cout << "." << endl;
  cout << "." << endl;
  cout << "PE_#15_ACC_=_" << Int48( acc2[15], acc1[15], acc0[15] ) << endl;
  cout << "PE_#16_ACC_=_" << Int48( acc2[16], acc1[16], acc0[16] ) << endl;
  cout << "PE_#17_ACC_=_" << Int48( acc2[17], acc1[17], acc0[17] ) << endl;
  cout << "." << endl;
  cout << "." << endl;
  cout << "PE_#31_ACC_=_" << Int48( acc2[31], acc1[31], acc0[31] ) << endl;
  cout << "PE_#32_ACC_=_" << Int48( acc2[32], acc1[32], acc0[32] ) << endl;
  cout << "PE_#33_ACC_=_" << Int48( acc2[33], acc1[33], acc0[33] ) << endl;
  cout << "." << endl;
  cout << "." << endl;
  cout << "PE_#47_ACC_=_" << Int48( acc2[47], acc1[47], acc0[47] ) << endl;
  cout << "PE_#48_ACC_=_" << Int48( acc2[48], acc1[48], acc0[48] ) << endl;
  cout << "PE_#49_ACC_=_" << Int48( acc2[49], acc1[49], acc0[49] ) << endl;
  cout << "." << endl;
  cout << "." << endl;
  cout << "PE_#N-1_ACC_=_" << Int48( acc2[numberOfPEs-2], acc1[numberOfPEs-2], acc0[numberOfPEs-2] ) << endl;
  cout << "PE_#N_ACC_=_" << Int48( acc2[numberOfPEs-1], acc1[numberOfPEs-1], acc0[numberOfPEs-1] ) << endl;

} // dumpAcc

// The following two operations are used to group parallel DSP-RAM operations

void DSPRAMClass::startInstructionDefn( int debugEn )
// verifies consistency of microoperations
{
    Int32 tempInt32_1, tempInt32_2;

//    if ( debugEn == 1 ) printRegisters();

    for ( int p=0; p < numberOfPEs; p++ ) {
        // By default, register contents will not change
        acc0Next[p] = acc0[p];
        acc1Next[p] = acc1[p];
        acc2Next[p] = acc2[p];
        shRegNext[p] = shReg[p];
        broadBusRegNext[p]  = broadBusReg[p];
        multInput1Next[p] = multInput1[p];
```

```
        multInput2Next[p] = multInput2[p];

        // The multiplier pipeline keeps on processing
        tempInt32_1 = multInput1[p];                    // Convert from Int16 to Int32
        tempInt32_2 = multInput2[p];
        multMidRegNext[p] = tempInt32_1 * tempInt32_2;  // All Int32 objects
        multOutputNext[p] = multMidReg[p];              // Both objects are Int32
    }
}

void DSPRAMClass::endInstructionDefn( int debugEn )
// executes instruction cycle
{
    if ( pushStackEn == true ) {
        stackPtr++;
        assert( stackPtr < hwStackSize );
        //cout << "Stack ptr is " << stackPtr << endl;
    } else if ( popStackEn == true ) {
        stackPtr--;
        assert( stackPtr >= 0 );
        //cout << "Stack ptr is " << stackPtr << endl;
    }

    for ( int p=0; p < numberOfPEs; p++ ) {
        acc0[p] = acc0Next[p];
        acc1[p] = acc1Next[p];
        acc2[p] = acc2Next[p];
        shReg[p] = shRegNext[p];
        if ( pushStackEn == true ) {
            ramEnStack[p][stackPtr] = ramEnNext[p];
        }
        broadBusReg[p] = broadBusRegNext[p];
        multInput1[p] = multInput1Next[p];
        multInput2[p] = multInput2Next[p];
        multMidReg[p] = multMidRegNext[p];
        multOutput[p] = multOutputNext[p];

        if ( toggleStackEn == true )
        {

          // Make sure something was pushed first
          assert( stackPtr > 0 );

          if ( ramEnStack[p][stackPtr] == true )
          {

            ramEnStack[p][stackPtr] = false;

          } // if
          else if ( ramEnStack[p][stackPtr-1] == true )
          {

            // Only toggle if previous was set.

            ramEnStack[p][stackPtr] = true;

          } // else if

        } // if

    }

    incrCycleCount();

    if ( debugEn == 1 ) printRegisters();

    if ( debugEn == 1 ) {
        cout << endl;
        cout << "Broadcast_bus___=_" << broadcastBusValue << endl
             << "Left_bus_value__=_" << leftBusValue << endl
             << "Right_bus_value_=_" << rightBusValue << endl;
    }
    pushStackEn = false;
    popStackEn = false;
    toggleStackEn = false;

}

void DSPRAMClass::incrCycleCount()
{
    cycleCount++;
}

const int DSPRAMClass::getCycleCount()
{

    return cycleCount;

} // getCycleCount

const int DSPRAMClass::getNumberOfPEs()
{
```

```
    return numberOfPEs;

} // getNumberOfPEs

const int DSPRAMClass::getRamBankSize()
{

    return ramBankSize;

} // getRamBankSize

const int DSPRAMClass::getHwStackSize()
{

    return hwStackSize;

} // getHwStackSize

void DSPRAMClass::enableByteSelect( int byte )
{

  byteSelectEnabled = true;
  useByte           = byte ? 8 : 0;

} // enableByteSelect

void DSPRAMClass::disableByteSelect()
{

  byteSelectEnabled = false;

} // disableByteSelect
```

# B.2 Integer Class

Listing B.3: Integer Classes Definition

```
//
//   Integer Class Declarations
//
//   Version 1.0   February 10, 2000
//           1.1   February 13, 2000
//           1.2   February 16, 2000
//           1.3   February 20, 2000
//           1.4   October  15, 2000   Added Int32 class.
//   Version 2.0   October  26, 2000
//           2.1   November 22, 2000
//
// $Id: integerClasses.h,v 1.1 2002/08/01 16:10:48 dillen Exp $
//
// $Log: integerClasses.h,v $
// Revision 1.1  2002/08/01 16:10:48  dillen
// Initial revision
//

#ifndef INTEGERCLASSES_H
#define INTEGERCLASSES_H

#include <iostream>
#include "shiftValueType.h"

#define MAX_INT16  32767
#define MIN_INT16 -32768

class Int32;
class Int48;

class Int16 {
    friend std::istream &operator>>( std::istream &, Int16 & );
    friend std::ostream &operator<<( std::ostream &, const Int16 & );
    friend class Int32;
    friend class Int48;
public:
    Int16( int = 0 );              // default constructor
    Int16( const Int16 & );        // copy constructor
    operator int() const;          // conversion from Int16 to int
    Int16 &setValue( int );
    Int16 &operator&( const Int16 & );
    Int16 &operator=( const Int16 & );
    bool  operator==( const Int16 & );
    bool  operator<( const Int16 & );
    bool  operator>( const Int16 & );
private:
    long int value;
};
```

```
class Int32 {
    friend std::ostream &operator<<( std::ostream &, const Int32 & );
    friend class Int16;
    friend class Int48;
public:
    Int32( int = 0 );                  // default constructor
    Int32( const Int32 & );            // copy constructor
    operator int() const;              // conversion from Int32 to int
    operator Int48() const;            // conversion from Int32 to Int48
    Int32 &setValue( int );
    Int32 operator+( const Int32 & );
    Int32 &operator=( const Int32 & );
    Int32 &operator=( const Int16 & );
private:
    long int value;
};

class Int48 {
    friend std::ostream &operator<<( std::ostream &, const Int48 & );
public:
    Int48( int = 0 );                  // default constructor
    Int48( const Int32 & );
    Int48( const Int16 &, const Int16 &, const Int16 & );
    Int48( const Int48 & );            // copy constructor
    // operator int() const;           // conversion from Int48 to int
    Int48 &setValue( int );
    Int48 &setValue( const Int16 &, const Int16 &, const Int16 & );
    Int48 &partition( Int16 &, Int16 &, Int16 & );
    Int48 &shift( ShiftValue );
    Int48 operator+( const Int48 & );
    Int48 &operator=( const Int48 & );
    Int48 &operator=( const Int32 & );
    Int48 &operator=( const Int16 & );
private:
    long int valueH, valueL;
};

#endif
```

## Listing B.4: Integer Classes Implementation

```
//
//   Integer Class Function Definitions
//
//   Version 1.0   February 16, 2000
//   Version 2.0   October  26, 2000
//           2.1   November 22, 2000    Added valueH and valueL to Int48
//
// $Id: integerClasses.cpp,v 1.1 2002/08/01 16:12:59 dillen Exp $
//
// $Log: integerClasses.cpp,v $
// Revision 1.1  2002/08/01 16:12:59  dillen
// Initial revision
//

#include "integerClasses.h"

// Class Int16 Functions

using std::istream;
using std::ostream;

istream &operator>>( istream &input, Int16 &rightInt16 )
{
    input >> rightInt16.value;
    rightInt16.value &= 0x0FFFF;
    if ( rightInt16.value & 0x00008000 )
    {

        rightInt16.value |= 0xFFFF0000;

    } // if

    return input;
}

ostream &operator<<( ostream &output, const Int16 &integer16 )
{
    output << integer16.value;
    return output;
}

Int16::Int16( int initialValue )
{
    value = initialValue & 0x0000FFFF;

    if ( value & 0x00008000 )
    {

        value |= 0xFFFF0000;
```

```
    } // if
}

Int16::Int16( const Int16 &originalInt16 )
{
    value = originalInt16.value;
    if ( value & 0x00008000 )
    {

        value |= 0xFFFF0000;

    } // if
}

Int16::operator int() const
// conversion from Int16 to int
{
    return value;
}

Int16 &Int16::setValue( int newValue )
{
    value = newValue & 0x0000FFFF;

    if ( value & 0x00008000 )
    {

        value |= 0xFFFF0000;

    } // if

    return *this;
}

Int16 &Int16::operator&( const Int16 &rightInt16 )
{
    value = value & rightInt16.value;
    return *this;
}

Int16 &Int16::operator=( const Int16 &rightInt16 )
{
    value = rightInt16.value;
    if ( value & 0x00008000 )
    {

        value |= 0xFFFF0000;

    } // if
    return *this;
}

bool Int16::operator==( const Int16 &rightInt16 )
{
    return ( value == rightInt16.value );
}

bool Int16::operator<( const Int16 &rightInt16 )
{
    return ( value < rightInt16.value );
}

bool Int16::operator>( const Int16 &rightInt16 )
{
    return ( value > rightInt16.value );
}


// Class Int32 Functions
//
//    Assumptions:
//       A long int is at least 32 bits wide, and may be exactly 32 bits wide.

ostream &operator<<( ostream &output, const Int32 &integer32 )
{
    output << integer32.value;
    return output;
}

Int32::Int32( int initialValue )
{
    value = initialValue & 0x00000000FFFFFFFF;

    if (value & 0x0000000080000000)
    {

        value |= 0xFFFFFFFF00000000;

    } // if
}
```

```
Int32::Int32( const Int32 &original )
{
    value = original.value;
}

Int32::operator int() const
// conversion from Int32 to int
{
    return value;
}

Int32::operator Int48() const
{
    return Int48( value );
}

Int32 &Int32::setValue( int newValue )
{
    value = newValue & 0x00000000FFFFFFFF;

    if ( value & 0x0000000080000000)
    {

        value |= 0xFFFFFFFF00000000;

    } // if

    return *this;
}

Int32 Int32::operator*( const Int32 &rightInt32 )
{

  return Int32( value * rightInt32.value );

}

Int32 &Int32::operator=( const Int32 &rightInt32 )
{
    value = rightInt32.value;
    return *this;
}

Int32 &Int32::operator=( const Int16 &rightInt16 )
{
    value = rightInt16.value;
    if ( value & 0x00008000 )
    {

        value |= 0xFFFF0000;

    } // if

    return *this;
}


// Class Int48 Functions
//
//    Assumptions:
//       A long int is at least 32 bits wide, and may be exactly 32 bits wide.

ostream &operator<<( ostream &output, const Int48 &integer48 )
{
    if ( integer48.valueH == 0 )
        output << integer48.valueL;
    else
    {

        // MUST 0 extend valueL for proper display ... SJD
        output.setf(std::ios::hex,std::ios::basefield);
        output << "$" << integer48.valueH << integer48.valueL;
        output.setf(std::ios::dec,std::ios::basefield);

    } // else

    return output;
}

Int48::Int48( int initialValue )
{
    valueL = initialValue;

    if ( valueL & 0x0000000080000000 )
    {

        valueH = -1;

    } // if
    else
    {
```

```
        valueH = 0;

    } // else

}

Int48::Int48( const Int32 &integer32 )
{
    valueL = integer32.value;

    if ( valueL & 0x0000000080000000 )
    {

        valueH = -1;

    } // if
    else
    {

        valueH = 0;

    } // else

}

Int48::Int48( const Int16 &word2, const Int16 &word1, const Int16 &word0 )
{
    valueH = word2;
    valueL = ((word1.value << 16) & 0xFFFF0000) | (word0.value & 0x0000FFFF);
}

Int48::Int48( const Int48 &original )
{
    valueH = original.valueH;
    valueL = original.valueL;
}

Int48 &Int48::setValue( int newValue )
{
    valueL = newValue;

    if ( valueL & 0x0000000080000000 )
    {

        valueH = -1;

    } // if
    else
    {

        valueH = 0;

    } // else

    return *this;
}

Int48 &Int48::setValue( const Int16 &word2, const Int16 &word1, const Int16 &word0 )
{
    valueH = word2;
    valueL = ((word1.value << 16) & 0xFFFF0000) | (word0.value & 0x0000FFFF);

    return *this;
}

Int48 &Int48::partition( Int16 &word2, Int16 &word1, Int16 &word0 )
{
    word2.value = valueH & 0x0FFFF;
    word1.value = (valueL >> 16) & 0x0FFFF;
    word0.value = valueL & 0x0FFFF;

    return *this;
}

Int48 &Int48::shift( ShiftValue sv )
{
    long int X;

    switch ( sv ) {
        case logicalShiftLeft:
            if ( (valueL & 0x080000000) == 0 ) {
                valueH = (valueH * 2) & 0x0FFFF;
                valueL = (valueL * 2) & 0x0FFFFFFFF;
            } else {
                valueH = ((valueH * 2) + 1) & 0x0FFFF;
                valueL = (valueL * 2) & 0x0FFFFFFFF;
            }
            break;

        case LSL8:
            X = valueL & 0x0FFFFFFFF;
            X = X >> 24;
```

```
        valueL = (valueL << 8);
        valueL = valueL & 0x0FFFFF00;
        valueH = (valueH << 8);
        valueH = valueH & 0x0FFFFF00;
        valueH |= X;
        break;

    case LSL16:
        X = valueL & 0x0FFFFFFF;
        X = X >> 16;
        valueL = (valueL << 16);
        valueL = valueL & 0x0FFF0000;
        valueH = (valueH << 16);
        valueH = valueH & 0x0FFF0000;
        valueH |= X;
        break;

    case LSL32:
        X = valueL & 0x00000FFF;
        valueH = X;
        valueL = 0;
        break;

    case noShift:
        break;

    case logicalShiftRight:
        if ( (valueH & 0x0001) == 0 ) {
            valueH = (valueH & 0x0FFF) / 2;
            valueL = (valueL / 2) & 0x07FFFFFF;
        } else {
            valueH = (valueH & 0x0FFF) / 2;
            valueL = ((valueL / 2) & 0x07FFFFFF) | 0x080000000;
        }
        break;

    case arithmeticShiftRight:
        if ( (valueH & 0x0001) == 0 ) {
            if ( valueH & 0x08000 == 0 )
                valueH = (valueH & 0x0FFF) / 2;
            else
                valueH = ((valueH & 0x0FFF) / 2) | 0x08000;
            valueL = (valueL / 2) & 0x07FFFFFF;
        } else {
            if ( valueH & 0x08000 == 0 )
                valueH = (valueH & 0x0FFF) / 2;
            else
                valueH = ((valueH & 0x0FFF) / 2) | 0x08000;
            valueL = ((valueL / 2) & 0x07FFFFFF) | 0x080000000;
        }
        break;

    case ASR8:
        X = valueH & 0x000FF;
        valueH = valueH >> 8;
        if (valueH & 0x00080 == 1)
            valueH |= 0x0FF00;
        else
            valueH &= 0x000FF;

        valueL = valueL >> 8;
        valueL &= 0x000FFFFFF;
        X = X << 24;
        valueL |= X;
        break;

    case ASR16:
        X = valueH & 0x0FFF;
        if (valueH & 0x08000 == 1)
            valueH |= 0x0FFFF;
        else
            valueH &= 0x00000;

        valueL = valueL >> 16;
        valueL &= 0x00000FFFF;
        X = X << 16;
        valueL |= X;
        break;

    case ASR32:
        X = valueH & 0x0FFFF;
        valueL = X;
        if (valueH & 0x08000 == 1)
        {
            valueH |= 0x0FFFF;
            valueL |= 0x0FFFF0000;
        }
        else
        {
            valueH &= 0x00000;
            valueL &= 0x00000FFFF;
        }
```

```
            break;
        }

        return *this;
    }

    Int48 Int48::operator+( const Int48 &rightInt48 )
    {
        Int48 dummy;
        Int16 left2, left1, left0, right2, right1, right0;
        long int sum2, sum1, sum0;

        dummy = *this;
        dummy.partition( left2, left1, left0 );

        dummy = rightInt48;
        dummy.partition( right2, right1, right0 );

        sum0 = left0 + right0;
        sum1 = ( (sum0 & 0x010000) == 0 ) ? 0 : 1;
        sum0 = sum0 & 0x0FFFF;

        sum1 = sum1 + left1 + right1;
        sum2 = ( (sum1 & 0x010000) == 0 ) ? 0 : 1;
        sum1 = sum1 & 0x0FFFF;

        sum2 = ( sum2 + left2 + right2 ) & 0x0FFFF;

        return Int48(sum2, sum1, sum0);
    }

    Int48 &Int48::operator=( const Int48 &integer48 )
    {
        valueH = integer48.valueH;
        valueL = integer48.valueL;

        return *this;
    }

    Int48 &Int48::operator=( const Int32 &integer32 )
    {
        valueL = integer32.value;

        if ( valueL & 0x0000000080000000 )
        {

            valueH = -1;

        } // if
        else
        {

            valueH = 0;

        } // else

        return *this;
    }

    Int48 &Int48::operator=( const Int16 &integer16 )
    {
        valueL = integer16.value;

        if ( valueL & 0x0000000080000000 )
        {

            valueH = -1;

        } // if
        else
        {

            valueH = 0;

        } // else

        return *this;
    }
```

## B.3 Shift Types

Listing B.5: Shift Types Definition

```
// $Id: shiftValueType.h,v 1.1 2002/08/01 16:10:48 dillen Exp $
//
// $Log: shiftValueType.h,v $
// Revision 1.1  2002/08/01 16:10:48  dillen
```

```
// Initial revision
//

#ifndef SHIFTVALUETYPEH
#define SHIFTVALUETYPEH

// Revision 2.0     October 26, 2000

enum ShiftValue { logicalShiftLeft , noShift , logicalShiftRight ,
                  arithmeticShiftRight , LSL8, LSL16, LSL32,
                  ASR8, ASR16, ASR32};

#endif
```

# Appendix C

# DSP-RAM VHDL Source Code

Listing C.1: DSP-RAM System

```
--
-- dspramSystem.vhd
--
-- Steve Dillen
-- Jun. 18, 2002
--
-- $Id: dspramSystem.vhd,v 1.5 2002/08/09 22:55:06 dillen Exp dillen $
--
-- $Log: dspramSystem.vhd,v $
-- Revision 1.5  2002/08/09 22:55:06  dillen
-- Final revison for version 1
--
-- Revision 1.4  2002/07/15 22:07:56  dillen
-- Changed others='Z' to "ZZZZZZZZZZZZZZZZ" for FPGA compiler 11
--
-- Revision 1.3  2002/07/15 17:10:54  dillen
-- Fixed and simulated all bugs
--
-- Revision 1.2  2002/07/10 21:25:30  dillen
-- Added generics and changed glue logic
--
-- Revision 1.1  2002/07/09 22:51:35  dillen
-- Initial revision
--
--

library IEEE;
use IEEE.std_logic_1164.all;

library LOGIC;
use LOGIC.logicPackage.all;

library MEMORY;
library PEARRAY;

--library UNISIM;

-- numberOfPEs = 2 ** numberOfPEDecodeBits

entity dspramSystem is

  generic ( dw    : integer := 16;
            sbw   : integer := 16;
            obits : integer := 1;
            nsv   : integer := 4;
            sv0   : integer := 1;
            sv1   : integer := 2;
            sv2   : integer := 4;
            sv3   : integer := 6;
            sv4   : integer := 8;
            sv5   : integer := 16;
            sv6   : integer := 24;
            sv7   : integer := 32;
            isr   : boolean := true;  -- Include shift register
            sd    : integer := 2;     -- Depth of the stack
            ndb   : integer := 2;     -- Number of PE Decode bits
            nPE   : integer := 4;     -- Number of PEs
            nra   : integer := 2;     -- Number of RAMs in bank A
            nla   : integer := 1;     -- Number of address lines in bank A
            nrb   : integer := 2;     -- Number of RAMs in bank B
            nlb   : integer := 1 );   -- Number of address lines in bank B

  port ( clock        : in    std_logic;
```

```
        nReset        : in    std_logic;
        addressA      : in    std_logic_vector( (8 + nla + ndb) - 1 downto 0 );
        addressB      : in    std_logic_vector( (8 + nlb + ndb) - 1 downto 0 );
        controlWord   : in    std_logic_vector( 49 downto 0 );
        dataA         : inout std_logic_vector( dw - 1 downto 0 );
        dataB         : inout std_logic_vector( dw - 1 downto 0 );
        leftBus       : inout std_logic_vector( sbw - 1 downto 0 );
        rightBus      : inout std_logic_vector( sbw - 1 downto 0 );
        broadcastBus  : in    std_logic_vector( dw - 1 downto 0 ) );

end entity dspramSystem;

architecture RTL of dspramSystem is

  component ram is

    generic ( nPE : integer := nPE;
              nr  : integer := nra;
              nl  : integer := nla );

    port ( clock       : in    std_logic;
           nReset      : in    std_logic;
           readWrite   : in    std_logic;
           enable      : in    std_logic_vector( nPE - 1 downto 0 );
           address     : in    std_logic_vector( 7 + nl downto 0 );
           data        : inout std_logic_vector( 16 * nPE - 1 downto 0 ) );

  end component ram;

  component peArray is

    generic ( dw    : integer := 16;
              sbw   : integer := 16;
              obits : integer := 1;
              nsv   : integer := 4;
              sv0   : integer := 32;
              sv1   : integer := 16;
              sv2   : integer := 8;
              sv3   : integer := 1;
              sv4   : integer := 2;
              sv5   : integer := 4;
              sv6   : integer := 6;
              sv7   : integer := 12;
              isr   : boolean := false;
              sd    : integer := 2;
              nPE   : integer := 16 );

    port ( clock         : in    std_logic;
           nReset        : in    std_logic;
           controlWord   : in    std_logic_vector( 43 downto 0 );
           ramDataA      : inout std_logic_vector( (dw * nPE) - 1 downto 0 );
           ramDataB      : inout std_logic_vector( (dw * nPE) - 1 downto 0 );
           leftPE        : inout std_logic_vector( sbw - 1 downto 0 );
           rightPE       : inout std_logic_vector( sbw - 1 downto 0 );
           broadcastBus  : in    std_logic_vector( dw - 1 downto 0 );
           ramEnable     : out   std_logic_vector( nPE - 1 downto 0 ) );

  end component peArray;

  -- Bank A Signals
  signal bankAEnables : std_logic_vector( nPE - 1 downto 0 );
  signal bankAData    : std_logic_vector( dw * (2 ** ndb) - 1 downto 0 );

  -- Bank B Signals
  signal bankBEnables : std_logic_vector( nPE - 1 downto 0 );
  signal bankBData    : std_logic_vector( dw * (2 ** ndb) - 1 downto 0 );

  -- Enable signals
  signal peRamEnables           : std_logic_vector( nPE - 1 downto 0 );
  signal sequentialRamEnablesA  : std_logic_vector( 2 ** ndb - 1 downto 0 );
  signal sequentialRamEnablesB  : std_logic_vector( 2 ** ndb - 1 downto 0 );
  signal sequentialEnablesA     : std_logic_vector( nPE - 1 downto 0 );
  signal sequentialEnablesB     : std_logic_vector( nPE - 1 downto 0 );

  signal peRamAWriteEnables     : std_logic_vector( nPE - 1 downto 0 );
  signal peRamBWriteEnables     : std_logic_vector( nPE - 1 downto 0 );

  signal dataAOut : std_logic_vector( dw - 1 downto 0 );
  signal dataBOut : std_logic_vector( dw - 1 downto 0 );

begin

  -- Bank A
  bankA : ram generic map ( nPE => nPE,
                            nr  => nra,
                            nl  => nla )

              port    map ( clock     => clock,
                            nReset    => nReset,
                            readWrite => controlWord( 44 ),
                            enable    => bankAEnables,
                            address   => addressA( (7 + nla + ndb) downto ndb ),
```

```
                        data        => bankAData( dw * nPE - 1 downto 0 ) );

-- Bank B
bankB : ram generic map ( nPE => nPE,
                          nr  => nrb,
                          nl  => nlb )

          port    map ( clock     => clock,
                        nReset    => nReset,
                        readWrite => controlWord( 45 ),
                        enable    => bankBEnables,
                        address   => addressB ( (7 + nlb + ndb) downto ndb ),
                        data      => bankBData ( dw * nPE - 1 downto 0 ) );

-- PE Array
pes : peArray generic map ( dw    => dw,
                            sbw   => sbw,
                            obits => obits,
                            nsv   => nsv,
                            sv0   => sv0,
                            sv1   => sv1,
                            sv2   => sv2,
                            sv3   => sv3,
                            sv4   => sv4,
                            sv5   => sv5,
                            sv6   => sv6,
                            sv7   => sv7,
                            isr   => isr,
                            sd    => sd,
                            nPE   => nPE )
              port map ( clock         => clock,
                         nReset        => nReset,
                         controlWord   => controlWord( 43 downto 0 ),
                         ramDataA      => bankAData( dw * nPE - 1 downto 0 ),
                         ramDataB      => bankBData( dw * nPE - 1 downto 0 ),
                         leftPE        => leftBus,
                         rightPE       => rightBus,
                         broadcastBus  => broadcastBus,
                         ramEnable     => peRamEnables );

-- Glue Logic for sequential memory access and SIMD memory access
--
-- Start with the sequential write enable generation
seqWE_A : decoderN generic map ( numberOfSelectBits => ndb )
                   port    map ( input  => addressA( ndb - 1 downto 0 ),
                                 output => sequentialRamEnablesA );

seqWE_B : decoderN generic map ( numberOfSelectBits => ndb )
                   port    map ( input  => addressB( ndb - 1 downto 0 ),
                                 output => sequentialRamEnablesB );

-- Generate RAM Enable Signals
writeEnables : for i in 0 to nPE - 1 generate

  peRamAWriteEnables(i) <= peRamEnables( i ) and controlWord( 46 );
  peRamBWriteEnables(i) <= peRamEnables( i ) and controlWord( 47 );

  bankAData( dw * (i + 1) - 1 downto dw * i ) <=
    dataA when ( controlWord( 48 ) = '1' and controlWord( 44 ) = '0' and
                 sequentialEnablesA (i) = '1' )
          else "ZZZZZZZZZZZZZZZZ";

  bankBData( dw * (i + 1) - 1 downto dw * i ) <=
    dataB when ( controlWord( 49 ) = '1' and controlWord( 45 ) = '0' and
                 sequentialEnablesB(i) = '1' )
          else "ZZZZZZZZZZZZZZZZ";

  sequentialEnablesA (i) <= sequentialRamEnablesA (i) and controlWord(46);
  sequentialEnablesB(i) <= sequentialRamEnablesB (i) and controlWord(47);

end generate writeEnables;

-- Select between sequential and SIMD address writing
bankAEnable : mux2 generic map( busWidth => nPE )
                   port    map( ip0  => peRamAWriteEnables,
                                ip1  => sequentialEnablesA,
                                sel  => controlWord( 48 ),
                                op   => bankAEnables );

-- Select between sequential and SIMD address writing
bankBEnable : mux2 generic map( busWidth => nPE )
                   port    map( ip0  => peRamBWriteEnables,
                                ip1  => sequentialEnablesB,
                                sel  => controlWord( 49 ),
                                op   => bankBEnables );

-- Mux out the output
dataAOutMux : muxN generic map ( busWidth => dw,
                                 selWidth => ndb )
                   port    map ( ip   => bankAData,
                                 sel  => addressA( ndb - 1 downto 0 ),
                                 op   => dataAOut );
```

```
  dataBOutMux : muxN generic map ( busWidth => dw,
                                   selWidth => ndb )
                    port    map ( ip       => bankBData,
                                  sel      => addressB( ndb - 1 downto 0 ),
                                  op       => dataBOut );

  dataA <= dataAOut when ( controlWord( 48 ) = '1' and controlWord( 44 ) = '1' )
                    else "ZZZZZZZZZZZZZZZZ";
  dataB <= dataBOut when ( controlWord( 49 ) = '1' and controlWord( 45 ) = '1' )
                    else "ZZZZZZZZZZZZZZZZ";

end architecture RTL;
```

## Listing C.2: PE Array

```
--
-- peArray.vhd
--
-- Steve Dillen
-- Jun. 18, 2002
--
-- $Id: peArray.vhd,v 1.5 2002/08/09 21:18:34 dillen Exp dillen $
--
-- $Log: peArray.vhd,v $
-- Revision 1.5  2002/08/09 21:18:34  dillen
-- Final release for version 1
--
-- Revision 1.4  2002/07/18 20:06:15  dillen
-- Added PE generics to the peArray
--
-- Revision 1.3  2002/07/17 17:32:09  dillen
-- Fixed tri-state shift bus
--
-- Revision 1.2  2002/07/15 21:50:38  dillen
-- Fixed the buses
--

library IEEE;
use IEEE.std_logic_1164.all;

library PE;

library LOGIC;
use LOGIC.logicPackage.all;

entity peArray is

  generic ( dw     : integer := 16;
            sbw    : integer := 16;
            obits  : integer := 1;
            nsv    : integer := 4;
            sv0    : integer := 32;
            sv1    : integer := 16;
            sv2    : integer := 8;
            sv3    : integer := 1;
            sv4    : integer := 2;
            sv5    : integer := 4;
            sv6    : integer := 6;
            sv7    : integer := 12;
            isr    : boolean := false;
            sd     : integer := 2;
            nPE    : integer := 16 );

  port ( clock         : in    std_logic;
         nReset        : in    std_logic;
         controlWord   : in    std_logic_vector( 43 downto 0 );
         ramDataA      : inout std_logic_vector( (dw * nPE) - 1 downto 0 );
         ramDataB      : inout std_logic_vector( (dw * nPE) - 1 downto 0 );
         leftPE        : inout std_logic_vector( sbw - 1 downto 0 );
         rightPE       : inout std_logic_vector( sbw - 1 downto 0 );
         broadcastBus  : in    std_logic_vector( dw - 1 downto 0 );
         ramEnable     : out   std_logic_vector( nPE - 1 downto 0 ) );

end peArray;

architecture RTL of peArray is

  component pe is

    generic ( dw     : integer := 16;
              sbw    : integer := 16;
              obits  : integer := 1;
              nsv    : integer := 4;
              sv0    : integer := 32;
              sv1    : integer := 16;
              sv2    : integer := 8;
              sv3    : integer := 1;
              sv4    : integer := 2;
              sv5    : integer := 4;
```

```
            sv6    : integer := 6;
            sv7    : integer := 12;
            isr    : boolean := false;
            sd     : integer := 2 );

    port ( clock        : in    std_logic;
           nReset       : in    std_logic;
           controlWord  : in    std_logic_vector( 43 downto 0 );
           ramDataA     : inout std_logic_vector( dw - 1 downto 0 );
           ramDataB     : inout std_logic_vector( dw - 1 downto 0 );
           leftPE       : inout std_logic_vector( sbw - 1 downto 0 );
           rightPE      : inout std_logic_vector( sbw - 1 downto 0 );
           broadcastBus : in    std_logic_vector( dw - 1 downto 0 );
           ramEnable    : out   std_logic );

  end component pe;

  signal interPEBus : std_logic_vector( ( sbw * (nPE + 1) ) - 1 downto 0 );
  signal nControl32 : std_logic;
  signal nControl35 : std_logic;

begin

  U0 : bidirectional generic map ( busWidth => sbw )
                     port    map ( internal  => interPEBus( sbw - 1 downto 0 ),
                                   direction => controlWord(35),
                                   external  => leftPE );

  U1 : bidirectional generic map ( busWidth => sbw )
                     port    map ( internal  => interPEBus( sbw * (nPE + 1) - 1
                                                  downto sbw * nPE ),
                                   direction => controlWord(36),
                                   external  => rightPE );

  -- Generate the array of PEs
  PEs : for i in 0 to nPE - 1 generate

    PEi : pe generic map ( dw    => dw,
                           sbw   => sbw,
                           obits => obits,
                           nsv   => nsv,
                           sv0   => sv0,
                           sv1   => sv1,
                           sv2   => sv2,
                           sv3   => sv3,
                           sv4   => sv4,
                           sv5   => sv5,
                           sv6   => sv6,
                           sv7   => sv7,
                           isr   => isr,
                           sd    => sd )

             port map ( clock        => clock,
                        nReset       => nReset,
                        controlWord  => controlWord,
                        ramDataA     => ramDataA( dw * (i + 1) - 1
                                          downto dw * i ),
                        ramDataB     => ramDataB( dw * (i + 1) - 1
                                          downto dw * i ),
                        leftPE       => interPEBus( sbw * (i + 1) - 1
                                          downto sbw * i ),
                        rightPE      => interPEBus( sbw * (i + 2) - 1
                                          downto sbw * (i + 1) ),
                        broadcastBus => broadcastBus,
                        ramEnable    => ramEnable( i ) );

  end generate PEs;

end RTL;
```

## Listing C.3: RAM Column

```
--
-- $Id: ramColumn.vhd,v 1.6 2002/08/21 19:36:30 dillen Exp $
--
-- $Log: ramColumn.vhd,v $
-- Revision 1.6  2002/08/21 19:36:30  dillen
-- Placed UNISIM library back in for simulation purposes
--
-- Revision 1.5  2002/08/21 17:52:58  dillen
-- Shortened generic parameter names so FPGA Compiler II would not
-- complain
--
-- Revision 1.4  2002/08/09 21:17:18  dillen
-- Final release for version 1
--
-- Revision 1.3  2002/07/17 17:30:26  dillen
-- Fixed bug for single ram column
--
-- Revision 1.2  2002/07/10 17:55:26  dillen
```

```
-- Added generics for number of RAM's and enable signals
--
-- Revision 1.1  2002/07/09 23:00:16  dillen
-- Initial revision
--
--

library IEEE;
use IEEE.std_logic_1164.all;

library UNISIM;

library LOGIC;
use LOGIC.logicPackage.all;

entity ramColumn is

  generic ( nr : integer := 2;
            nl : integer := 1 );

  port ( clock     : in    std_logic;
         reset     : in    std_logic;
         readWrite : in    std_logic;
         enable    : in    std_logic;
         address   : in    std_logic_vector( 7 + nl downto 0 );
         data      : inout std_logic_vector( 15 downto 0 ) );

end entity ramColumn;

architecture RTL of ramColumn is

  component RAMB4_S16 is

    port ( we   : in  std_logic;
           en   : in  std_logic;
           rst  : in  std_logic;
           clk  : in  std_logic;
           addr : in  std_logic_vector( 7 downto 0 );
           di   : in  std_logic_vector( 15 downto 0 );
           do   : out std_logic_vector( 15 downto 0 ) );

  end component RAMB4_S16;

  -- Data bus signals for tri-stating
  signal dataIn         : std_logic_vector( 15 downto 0 );
  signal dataOut        : std_logic_vector( (16 * (2 ** nl)) - 1 downto 0 );
  signal dataOutSingle  : std_logic_vector( 15 downto 0 );

  signal writeEnable : std_logic;

  signal ramEnable : std_logic_vector( nr - 1 downto 0 );
  signal ramDecode : std_logic_vector( 2 ** nl - 1 downto 0 );

begin

  writeEnable <= not readWrite;

  singleRAM : if nr = 1 generate

    ram : RAMB4_S16 port map ( we   => writeEnable,
                               en   => enable,
                               rst  => reset,
                               clk  => clock,
                               addr => address( 7 downto 0 ),
                               di   => dataIn,
                               do   => dataOutSingle );

  end generate singleRAM;

  multipleRAMs : if nr > 1 and nl > 0 generate

    -- Allocate the decoder
    decoder : decoderN generic map ( numberOfSelectBits => nl )
                       port    map ( input  => address( 7 + nl downto 8 ),
                                     output => ramDecode );

    -- Allocate the memory banks
    RAMs : for i in 0 to nr - 1 generate

      ramEnable(i) <= ramDecode(i) and enable;

      rami : RAMB4_S16 port map ( we   => writeEnable,
                                  en   => ramEnable( i ),
                                  rst  => reset,
                                  clk  => clock,
                                  addr => address( 7 downto 0 ),
                                  di   => dataIn,
                                  do   => dataOut( 16*(i+1) - 1 downto 16*i ));

    end generate RAMs;

    -- Mux out the dataout signals
```

```
    mux : muxN generic map ( busWidth => 16,
                             selWidth => nl )
                port    map ( ip       => dataOut,
                              sel      => address( 7 + nl downto 8 ),
                              op       => dataOutSingle );

  end generate multipleRAMs;

  U0 : tri generic map ( busWidth => 16 )
           port    map ( input    => data,
                         enable   => writeEnable,
                         output   => dataIn );

  U1 : tri generic map ( busWidth => 16 )
           port    map ( input    => dataOutSingle,
                         enable   => readWrite,
                         output   => data );

end architecture RTL;
```

## Listing C.4: RAM

```
--
-- ram.vhd
--
-- Steve Dillen
-- Jun. 18, 2002
--
-- $Id: ram.vhd,v 1.3 2002/08/21 17:52:58 dillen Exp $
--
-- $Log: ram.vhd,v $
-- Revision 1.3  2002/08/21 17:52:58  dillen
-- Shortened generic parameter names so FPGA Compiler II would not
-- complain
--
-- Revision 1.2  2002/07/10 21:22:09  dillen
-- Added generics and control signals
--
-- Revision 1.1  2002/07/09 23:00:43  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

entity ram is

  generic ( nPE : integer := 16;
            nr  : integer := 2;
            nl  : integer := 1 );

  port ( clock     : in    std_logic;
         nReset    : in    std_logic;
         readWrite : in    std_logic;
         enable    : in    std_logic_vector( nPE - 1 downto 0 );
         address   : in    std_logic_vector( 7 + nl downto 0 );
         data      : inout std_logic_vector( 16 * nPE - 1 downto 0 ) );

end ram;

architecture RTL of ram is

  component ramColumn is

    generic ( nr : integer := 2;
              nl : integer := 1 );

    port ( clock     : in    std_logic;
           reset     : in    std_logic;
           readWrite : in    std_logic;
           enable    : in    std_logic;
           address   : in    std_logic_vector( 7 + nl downto 0 );
           data      : inout std_logic_vector( 15 downto 0 ) );

  end component ramColumn;

  signal reset : std_logic;

begin

  reset <= not nReset;

  RAMs : for i in 0 to nPE - 1 generate

    RAMi : ramColumn generic map ( nr => nr,
                                   nl => nl )
                     port map ( clock     => clock,
                                reset     => reset,
                                readWrite => readWrite,
                                enable    => enable(i),
```

```
                                address   => address,
                                data      => data( 16*(j+1) - 1 downto 16*j ));

  end generate RAMs;

end RTL;
```

## Listing C.5: Controller

```
--
-- dspramController.vhd
--
-- Jul 1, 2002
--
-- $Id: dspramController.vhd,v 1.4 2002/08/09 21:16:12 dillen Exp dillen $
--
-- $Log: dspramController.vhd,v $
-- Revision 1.4  2002/08/09 21:16:12  dillen
-- Final release for version 1
--
-- Revision 1.3  2002/07/17 17:22:46  dillen
-- Fixed problems with tri-state buffers
--
-- Revision 1.2  2002/07/15 21:42:56  dillen
-- Changed to inout bus architecture
--
-- Revision 1.1  2002/07/15 17:25:17  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

library SYSTEM;

library LOGIC;
use LOGIC.logicPackage.all;

entity dspramController is

  generic ( dw    : integer := 16;
            sbw   : integer := 35;
            obits : integer := 1;
            nsv   : integer := 4;
            sv0   : integer := 32;
            sv1   : integer := 16;
            sv2   : integer := 8;
            sv3   : integer := 1;
            sv4   : integer := 2;
            sv5   : integer := 4;
            sv6   : integer := 6;
            sv7   : integer := 8;
            isr   : boolean := false;
            sd    : integer := 2;
            ndb   : integer := 5;
            nPE   : integer := 20;
            nra   : integer := 2;
            nla   : integer := 1;
            nrb   : integer := 2;
            nlb   : integer := 1 );

  port ( clock        : in  std_logic;
         nReset       : in  std_logic;
         chipEnable   : in  std_logic;
         APBSelect    : in  std_logic;
         writeEnable  : in  std_logic;
         writeData    : in  std_logic_vector( 31 downto 0 );
         address      : in  std_logic_vector( 6 downto 2 );
         readData     : out std_logic_vector( 31 downto 0 ) );

end entity dspramController;

architecture ARM of dspramController is

  component dspramSystem is

    generic ( dw    : integer := 16;
              sbw   : integer := 48;
              obits : integer := 1;
              nsv   : integer := 4;
              sv0   : integer := 1;
              sv1   : integer := 2;
              sv2   : integer := 4;
              sv3   : integer := 6;
              sv4   : integer := 8;
              sv5   : integer := 16;
              sv6   : integer := 24;
              sv7   : integer := 32;
              isr   : boolean := false;
              sd    : integer := 2;
```

```
            ndb    : integer := 4;
            nPE    : integer := 16;
            nra    : integer := 2;
            nla    : integer := 1;
            nrb    : integer := 2;
            nlb    : integer := 1 );

  port ( clock       : in    std_logic;
         nReset      : in    std_logic;
         addressA    : in    std_logic_vector( (8+nla+ndb) - 1 downto 0 );
         addressB    : in    std_logic_vector( (8+nlb+ndb) - 1 downto 0 );
         controlWord : in    std_logic_vector( 49 downto 0 );
         dataA       : inout std_logic_vector( dw - 1  downto 0 );
         dataB       : inout std_logic_vector( dw - 1  downto 0 );
         leftBus     : inout std_logic_vector( sbw - 1 downto 0 );
         rightBus    : inout std_logic_vector( sbw - 1 downto 0 );
         broadcastBus : in   std_logic_vector( dw - 1  downto 0 ) );

end component dspramSystem;

-- Define Register Addresses
constant ADDRESS_A_REGISTER : std_logic_vector( 6 downto 2 ) := "00000";
constant ADDRESS_B_REGISTER : std_logic_vector( 6 downto 2 ) := "00001";
constant DATA_A_REGISTER    : std_logic_vector( 6 downto 2 ) := "00010";
constant DATA_B_REGISTER    : std_logic_vector( 6 downto 2 ) := "00011";
constant CONTROL_WORD_LSW   : std_logic_vector( 6 downto 2 ) := "00100";
constant CONTROL_WORD_MSW   : std_logic_vector( 6 downto 2 ) := "00101";
constant LEFT_BUS0          : std_logic_vector( 6 downto 2 ) := "00110";
constant LEFT_BUS1          : std_logic_vector( 6 downto 2 ) := "00111";
constant LEFT_BUS2          : std_logic_vector( 6 downto 2 ) := "01000";
constant RIGHT_BUS0         : std_logic_vector( 6 downto 2 ) := "01001";
constant RIGHT_BUS1         : std_logic_vector( 6 downto 2 ) := "01010";
constant RIGHT_BUS2         : std_logic_vector( 6 downto 2 ) := "01011";
constant BROADCAST_BUS      : std_logic_vector( 6 downto 2 ) := "01100";
constant CONTROL_REGISTER   : std_logic_vector( 6 downto 2 ) := "01101";

-- Define Control Register Bits
constant CLOCK_BIT : integer := 0;
constant RESET_BIT : integer := 15;

signal addressA         : std_logic_vector( 15 downto 0 );
signal addressB         : std_logic_vector( 15 downto 0 );
signal dataAIn          : std_logic_vector( dw - 1 downto 0 );
signal dataBIn          : std_logic_vector( dw - 1 downto 0 );
signal controlWord      : std_logic_vector( 63 downto 0 );
signal leftBusIn        : std_logic_vector( sbw - 1 downto 0 );
signal rightBusIn       : std_logic_vector( sbw - 1 downto 0 );
signal broadcastBus     : std_logic_vector( dw - 1 downto 0 );
signal controlRegister  : std_logic_vector( 15 downto 0 );
signal leftBusOut       : std_logic_vector( sbw - 1 downto 0 );
signal rightBusOut      : std_logic_vector( sbw - 1 downto 0 );
signal dataAOut         : std_logic_vector( dw - 1 downto 0 );
signal dataBOut         : std_logic_vector( dw - 1 downto 0 );

signal dataASystem      : std_logic_vector( dw - 1 downto 0 );
signal dataBSystem      : std_logic_vector( dw - 1 downto 0 );
signal leftBusSystem    : std_logic_vector( sbw - 1 downto 0 );
signal rightBusSystem   : std_logic_vector( sbw - 1 downto 0 );

signal nControl35   : std_logic;
signal nControl36   : std_logic;
signal dataAControl : std_logic;
signal nDataAControl : std_logic;
signal dataBControl : std_logic;
signal nDataBControl : std_logic;

begin

nControl35   <= not controlWord(35);
nControl36   <= not controlWord(36);

dataAControl  <= controlWord(48) and controlWord(46) and controlWord(44);
nDataAControl <= not dataAControl;

dataBControl  <= controlWord(49) and controlWord(47) and controlWord(45);
nDataBControl <= not dataAControl;

shiftbus1 : if sbw = dw generate

  writeRegisters : process( nReset, clock ) is
  begin

    if ( nReset = '0' ) then                    -- Asynchronous Reset

      addressA    <= ( others => '0' );
      addressB    <= ( others => '0' );
      dataAIn     <= ( others => '0' );
      dataBIn     <= ( others => '0' );
      controlWord <= ( others => '0' );
      leftBusIn   <= ( others => '0' );
      rightBusIn  <= ( others => '0' );
      broadcastBus <= ( others => '0' );
```

```
      controlRegister <= ( others => '0' );

    elsif ( clock'event and clock = '1' ) then  -- Rising edge of clock

      if ( ( APBSelect and writeEnable and chipEnable ) = '1' ) then

        case address is

          when ADDRESS_A_REGISTER => addressA <= writeData( 15 downto 0 );
          when ADDRESS_B_REGISTER => addressB <= writeData( 15 downto 0 );
          when DATA_A_REGISTER    => dataAIn  <= writeData( dw - 1 downto 0 );
          when DATA_B_REGISTER    => dataBIn  <= writeData( dw - 1 downto 0 );
          when CONTROL_WORD_LSW   => controlWord( 31 downto 0 )  <= writeData;
          when CONTROL_WORD_MSW   => controlWord( 63 downto 32 ) <= writeData;
          when LEFT_BUS0          => leftBusIn( dw - 1 downto  0 ) <= writeData( dw - 1 downto 0 );

          when RIGHT_BUS0         => rightBusIn( dw - 1 downto  0 ) <= writeData( dw - 1 downto 0 );

          when BROADCAST_BUS      => broadcastBus <= writeData( dw - 1 downto 0 );
          when CONTROL_REGISTER   => controlRegister <= writeData( 15 downto 0 );
          when others             => null;

        end case;

      end if;

    end if;

  end process writeRegisters;

  readRegisters : process( address, addressA, addressB, dataAOut, dataBOut,
                           controlWord, leftBusOut, rightBusOut, broadcastBus,
                           controlRegister ) is
  begin

    readData <= ( others => '0' );

    case address is

      when ADDRESS_A_REGISTER => readData( 15 downto 0 ) <= addressA;
      when ADDRESS_B_REGISTER => readData( 15 downto 0 ) <= addressB;
      when DATA_A_REGISTER    => readData( dw - 1 downto 0 ) <= dataAOut;
      when DATA_B_REGISTER    => readData( dw - 1 downto 0 ) <= dataBOut;
      when CONTROL_WORD_LSW   => readData <= controlWord( 31 downto 0 );
      when CONTROL_WORD_MSW   => readData <= controlWord( 63 downto 32 );
      when LEFT_BUS0          => readData( dw - 1 downto 0 ) <=
                                   leftBusOut( dw - 1 downto 0 );
      when RIGHT_BUS0         => readData( dw - 1 downto 0 ) <=
                                   rightBusOut( dw - 1 downto  0 );
      when BROADCAST_BUS      => readData( dw - 1 downto 0 ) <= broadcastBus;
      when CONTROL_REGISTER   => readData( 15 downto 0 ) <= controlRegister;

      when others             => null;

    end case;

  end process readRegisters;

end generate shiftBus1;

shiftbus3 : if sbw = dw * 3 generate

  writeRegisters : process( nReset, clock ) is
  begin

    if ( nReset = '0' ) then                    -- Asynchronous Reset

      addressA      <= ( others => '0' );
      addressB      <= ( others => '0' );
      dataAIn       <= ( others => '0' );
      dataBIn       <= ( others => '0' );
      controlWord   <= ( others => '0' );
      leftBusIn     <= ( others => '0' );
      rightBusIn    <= ( others => '0' );
      broadcastBus  <= ( others => '0' );
      controlRegister <= ( others => '0' );

    elsif ( clock'event and clock = '1' ) then  -- Rising edge of clock

      if ( ( APBSelect and writeEnable and chipEnable ) = '1' ) then

        case address is

          when ADDRESS_A_REGISTER => addressA <= writeData( 15 downto 0 );
          when ADDRESS_B_REGISTER => addressB <= writeData( 15 downto 0 );
          when DATA_A_REGISTER    => dataAIn  <= writeData( dw - 1 downto 0 );
          when DATA_B_REGISTER    => dataBIn  <= writeData( dw - 1 downto 0 );
          when CONTROL_WORD_LSW   => controlWord( 31 downto 0 )  <= writeData;
          when CONTROL_WORD_MSW   => controlWord( 63 downto 32 ) <= writeData;
          when LEFT_BUS0          => leftBusIn( dw - 1 downto  0 ) <=
                                       writeData( dw - 1 downto 0 );
          when LEFT_BUS1          => leftBusIn( dw * 2 - 1 downto dw ) <=
```

```
                                    writeData( dw - 1 downto 0 );
          when LEFT_BUS2          => leftBusIn( dw * 3 - 1 downto dw * 2 ) <=
                                    writeData( dw - 1 downto 0 );

          when RIGHT_BUS0         => rightBusIn( dw - 1 downto 0 ) <=
                                    writeData( dw - 1 downto 0 );
          when RIGHT_BUS1         => rightBusIn( dw * 2 - 1 downto dw ) <=
                                    writeData( dw - 1 downto 0 );
          when RIGHT_BUS2         => rightBusIn( dw * 3 - 1 downto dw * 2 ) <=
                                    writeData( dw - 1 downto 0 );

          when BROADCAST_BUS      => broadcastBus <=
                                    writeData( dw - 1 downto 0 );
          when CONTROL_REGISTER   => controlRegister <=
                                    writeData( 15 downto 0 );
          when others             => null;

        end case;

      end if;

    end if;

  end process writeRegisters;

  readRegisters : process( address, addressA, addressB, dataAOut, dataBOut,
                           controlWord, leftBusOut, rightBusOut, broadcastBus,
                           controlRegister ) is
  begin

    readData <= ( others => '0' );

    case address is

      when ADDRESS_A_REGISTER => readData( 15 downto 0 ) <= addressA;
      when ADDRESS_B_REGISTER => readData( 15 downto 0 ) <= addressB;
      when DATA_A_REGISTER    => readData( dw - 1 downto 0 ) <= dataAOut;
      when DATA_B_REGISTER    => readData( dw - 1 downto 0 ) <= dataBOut;
      when CONTROL_WORD_LSW   => readData <= controlWord( 31 downto 0 );
      when CONTROL_WORD_MSW   => readData <= controlWord( 63 downto 32 );
      when LEFT_BUS0          => readData( dw - 1 downto 0 ) <=
                                    leftBusOut( dw - 1 downto 0 );
      when LEFT_BUS1          => readData( dw - 1 downto 0 ) <=
                                    leftBusOut( dw * 2 - 1 downto dw );
      when LEFT_BUS2          => readData( dw - 1 downto 0 ) <=
                                    leftBusOut( dw * 3 - 1 downto dw * 2 );
      when RIGHT_BUS0         => readData( dw - 1 downto 0 ) <=
                                    rightBusOut( dw - 1 downto 0 );
      when RIGHT_BUS1         => readData( dw - 1 downto 0 ) <=
                                    rightBusOut( dw * 2 - 1 downto dw );
      when RIGHT_BUS2         => readData( dw - 1 downto 0 ) <=
                                    rightBusOut( dw * 3 - 1 downto dw * 2 );
      when BROADCAST_BUS      => readData( dw - 1 downto 0 ) <= broadcastBus;
      when CONTROL_REGISTER   => readData( 15 downto 0 ) <= controlRegister;

      when others             => null;

    end case;

  end process readRegisters;

end generate shiftBus3;

U0 : tri generic map ( busWidth => sbw )
         port    map ( input    => leftBusSystem,
                       enable   => controlWord(35),
                       output   => leftBusOut );

U1 : tri generic map ( busWidth => sbw )
         port    map ( input    => leftBusIn,
                       enable   => nControl35,
                       output   => leftBusSystem );

U2 : tri generic map ( busWidth => sbw )
         port    map ( input    => rightBusSystem,
                       enable   => controlWord(36),
                       output   => rightBusOut );

U3 : tri generic map ( busWidth => sbw )
         port    map ( input    => rightBusIn,
                       enable   => nControl36,
                       output   => rightBusSystem );

U4 : tri generic map ( busWidth => dw )
         port    map ( input    => dataASystem,
                       enable   => dataAControl,
                       output   => dataAOut );

U5 : tri generic map ( busWidth => dw )
         port    map ( input    => dataAIn,
                       enable   => nDataAControl,
                       output   => dataASystem );
```

```
U6 : tri generic map ( busWidth => dw )
       port    map ( input    => dataBSystem,
                     enable   => dataBControl,
                     output   => dataBOut );

U7 : tri generic map ( busWidth => dw )
       port    map ( input    => dataBIn,
                     enable   => nDataBControl,
                     output   => dataBSystem );

-- Map the DSP-RAM System
dspram : dspramSystem generic map ( dw    => dw,
                                    sbw   => sbw,
                                    obits => obits,
                                    nsv   => nsv,
                                    sv0   => sv0,
                                    sv1   => sv1,
                                    sv2   => sv2,
                                    sv3   => sv3,
                                    sv4   => sv4,
                                    sv5   => sv5,
                                    sv6   => sv6,
                                    sv7   => sv7,
                                    isr   => isr,
                                    sd    => sd,
                                    ndb   => ndb,
                                    nPE   => nPE,
                                    nra   => nra,
                                    nla   => nla,
                                    nrb   => nrb,
                                    nlb   => nlb )

                          port map ( clock       => controlRegister( CLOCK_BIT ),
                                     nReset      => controlRegister( RESET_BIT ),
                                     addressA    => addressA( (8 + nla + ndb) - 1 downto 0 ),
                                     addressB    => addressB( (8 + nlb + ndb ) - 1 downto 0 ),
                                     dataA       => dataASystem,
                                     dataB       => dataBSystem,
                                     controlWord => controlWord( 49 downto 0 ),
                                     leftBus     => leftBusSystem,
                                     rightBus    => rightBusSystem,
                                     broadcastBus => broadcastBus );

end architecture ARM;
```

## C.1 AMBA Protocol

The code that implements the AMBA protocol was provided by CMC for use with the rapid-prototyping platform [b:cmc]. The DSP-RAM controller was implemented as an AMBA Peripheral Bus component and was inserted into the code.

Listing C.6: AHB to APB Interface

```
------------------------------------------------------------------------
-- This confidential and proprietary software may be used only as
-- authorised by a licensing agreement from ARM Limited
--   (C) COPYRIGHT 2000 ARM Limited
--       ALL RIGHTS RESERVED
-- The entire notice above must be reproduced on all authorised
-- copies and copies may only be made to the extent permitted
-- by a licensing agreement from ARM Limited.
--
------------------------------------------------------------------------
-- Version and Release Control Information:
--
-- File Name            : $RCSfile: AHB2APB.vhd,v $
-- File Revision        : $Revision: 1.1 $
--
-- Release Information  : $State: Exp $
--
------------------------------------------------------------------------
-- Purpose :
--           Bridge component to connect APB system to AHB buses.
--
------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

------------------------------------------------------------------------
```

```
entity AHB2APB is
  port (
-- Inputs
        HCLK            : in    std_logic; -- system bus clock
        HRESETn         : in    std_logic; -- reset input (active low)
        HTRANS          : in    std_logic_vector(1 downto 0);
                                -- AHB transfer type
        HWRITE          : in    std_logic; -- AHB hwrite
        HSELAHBAPB      : in    std_logic; -- AHB peripheral select
        HREADYIn        : in    std_logic; -- AHB ready input
        PRDATA          : in    std_logic_vector(31 downto 0);
                                -- APB read data bus
        HWDATA          : in    std_logic_vector(31 downto 0);
                                -- AHB write data bus
        HADDR           : in    std_logic_vector(31 downto 0);
                                -- AHB address bus
-- Outputs
        PENABLE         : out   std_logic; -- APB peripheral enable
        HREADYOut       : out   std_logic; -- AHB ready output to S->M mux
        HRESP           : out   std_logic_vector(1 downto 0);
                                -- AHB response
        PWRITE          : out   std_logic; -- Peripheral bus Write
        PSELREGS        : out   std_logic; -- Peripheral select - register
                                -- peripheral
        PSELINTC        : out   std_logic; -- Peripheral select - interrupt
                                -- controller
        PWDATA          : out   std_logic_vector(31 downto 0);
                                -- APB write data bus
        HRDATA          : out   std_logic_vector(31 downto 0);
                                -- AHB read data bus
        PADDR           : out   std_logic_vector(31 downto 0)
                                -- APB address bus

        );
end AHB2APB;

------------------------------------------------------------------------
--
--                              AHB2APB
--                              =======
--
------------------------------------------------------------------------
--
-- Overview
-- ========
-- Bridge component to connect APB system to AHB buses
-- Provides chip select lines and enable signal to all APB peripherals.
--
------------------------------------------------------------------------

----============================ARCHITECTURE==============================--

architecture synth of AHB2APB is

------------------------------------------------------------------------
-- Component declarations
------------------------------------------------------------------------

------------------------------------------------------------------------
-- Constant declarations
------------------------------------------------------------------------
-- This constant defines the size of the peripheral address bus:
constant PADDRWIDTH     : integer  := 16;

-- APBIF states:
constant ST_IDLE        : std_logic_vector(2 downto 0) := "000";
constant ST_WWAIT       : std_logic_vector(2 downto 0) := "001";
constant ST_WRITE       : std_logic_vector(2 downto 0) := "010";
constant ST_READ        : std_logic_vector(2 downto 0) := "011";
constant ST_ENABLE      : std_logic_vector(2 downto 0) := "100";

-- constants for APB address decoding
constant INTCBASE       : std_logic_vector(27 downto 24) := "0001";
constant REGSBASE       : std_logic_vector(27 downto 24) := "0000";

-- HTRANS transfer type signal encoding
constant TRN_IDLE       : std_logic_vector(1 downto 0) := "00";
constant TRN_BUSY       : std_logic_vector(1 downto 0) := "01";
constant TRN_NONSEQ     : std_logic_vector(1 downto 0) := "10";
constant TRN_SEQ        : std_logic_vector(1 downto 0) := "11";

-- HRESP transfer response signal encoding
constant RSP_OKAY       : std_logic_vector(1 downto 0) := "00";
constant RSP_ERROR      : std_logic_vector(1 downto 0) := "01";
constant RSP_RETRY      : std_logic_vector(1 downto 0) := "10";
constant RSP_SPLIT      : std_logic_vector(1 downto 0) := "11";

------------------------------------------------------------------------
-- Signal declarations
------------------------------------------------------------------------

signal HSelReg          : std_logic;
-- HSELAHBAPB register
```

```
signal Valid            : std_logic;
-- Module is selected with Valid transfer

signal RegValid         : std_logic;
-- Module was selected with valid transfer

signal RegValidPrev     : std_logic;
-- Previous RegValid value

signal AddrCRegEn       : std_logic;
-- Enable for address and control registers

signal HAddrReg         : std_logic_vector(31 downto 0);
-- HADDR register

signal HTransReg        : std_logic_vector(1 downto 0);
-- HTRANS register

signal HWriteReg        : std_logic;
-- HWRITE register

signal HAddrMux         : std_logic_vector(31 downto 0);
-- HADDR multiplexor

signal PSELREGSInt      : std_logic;
-- Internal PSELREGS

signal PSELINTCInt      : std_logic;
-- Internal PSELINTC

signal WrBurst          : std_logic;
-- Write burst detection

signal WrBurstReg       : std_logic;
-- Write burst register

signal Wr2Rd            : std_logic;
-- Write to read detection

signal Wr2RdReg         : std_logic;
-- Write to read detection register

signal NextState        : std_logic_vector(2 downto 0);
-- State machine

signal CurrentState     : std_logic_vector(2 downto 0);
-- Current state

signal ApbEn            : std_logic;
-- High if in either READ or WRITE state

signal HReadyNext       : std_logic;
-- HREADYOut register input

signal iHREADYOut       : std_logic;
-- HREADYOut register

signal PWDATAEn         : std_logic;
-- PWDATA Register enable

signal iPENABLE         : std_logic;
-- Internal PENABLE

signal PSelRes          : std_logic;
-- PSEL reset

signal iPSELREGS        : std_logic;
-- Internal PSEL outputs

signal iPSELINTC        : std_logic;
-- Internal PSEL outputs

signal NextPWRITE       : std_logic;
-- PWRITE register input

signal iPADDR           : std_logic_vector(PADDRWIDTH - 1 downto 0);
-- Registered internal PADDR

signal iPWRITE          : std_logic;
-- Registered internal PWRITE

--------------------------------------------------------------------------------
-- Function declarations
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
--
-- Main body of code
-- ==================
--
--------------------------------------------------------------------------------
```

```
begin

-------------------------------------------------------------------------------
-- Valid transfer detection
-- The slave must only respond to a valid transfer, so this must be detected.
-------------------------------------------------------------------------------
p_HSelRegSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    HSelReg    <= '0';
  elsif (HCLK'event and HCLK = '1') then
    if (HREADYIn = '1') then
      HSelReg <= HSELAHBAPB;
    end if;
  end if;
end process p_HSelRegSeq;

-- Valid AHB transfers only take place when a non-sequential or sequential
-- transfer is shown on HTRANS - an idle or busy transfer should be ignored.
Valid     <= '1' when ((HSELAHBAPB = '1') and (HREADYIn = '1') and
                       ((HTRANS = TRN_NONSEQ) or (HTRANS = TRN_SEQ)))
          else
             '0';

RegValid <= '1' when ((HSelReg = '1') and
                      ((HTransReg = TRN_NONSEQ) or
                       (HTransReg = TRN_SEQ)))
          else
             '0';

-------------------------------------------------------------------------------
-- This register is used in the detection of a sequence of transfers where
-- there is an APB transfer, followed by a non-APB transfer and another APB
-- transfer. HREADYIn is not used as an enable to avoid extra APB cycles from
-- being inserted.
-------------------------------------------------------------------------------
p_RegValidPrevSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    RegValidPrev <= '0';
  elsif (HCLK'event and HCLK = '1') then
    RegValidPrev <= RegValid;
  end if;
end process p_RegValidPrevSeq;

AddrCRegEn <= HSELAHBAPB and HREADYIn;

-------------------------------------------------------------------------------
-- Address and control registers
-- Registers are used to store the address and control signals from the address
-- phase for use in the data phase of the transfer.
-- Only enabled when the HREADYIn input is HIGH and the module is addressed.
-------------------------------------------------------------------------------
p_AddrCntRegSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    HAddrReg     <= (others => '0');
    HTransReg    <= (others => '0');
    HWriteReg    <= '0';
  elsif (HCLK'event and HCLK = '1') then
    if (AddrCRegEn = '1') then
      HAddrReg   <= HADDR;
      HTransReg  <= HTRANS;
      HWriteReg  <= HWRITE;
    end if;
  end if;
end process p_AddrCntRegSeq;

-- Internally, the address used depends on the current cycle being a read or a
-- write, due to the different timing for each type of transfer.
-- As a write transfer does not start until the write data has been driven onto
-- HWDATA, HADDR will contain the address for the next transfer, so the
-- address for a write transfer is always pipelined. The HAddrReg registers
-- are used as the pipeline, as they will always hold the previous AHB
-- address.
-- So, a multiplexor is needed to select the address source depending on the
-- current transfer type:
-- HADDR for a read cycle.
-- HAddrReg for a write cycle.
-- HAddrReg is also selected when a non-sequential read (or write) is performed
-- when the APB has just started a write transfer, as by the time the current
-- APB cycle has finished, the AHB address will have moved onto the next
-- transfer.
-- HAddrReg is used as the default state as it will change less often than the
-- HADDR input.
--
-- HAddrMux is used to generate the PSELxInt signals and PADDR.

HAddrMux <= HAddrReg when ((RegValid = '1') and (RegValidPrev = '0') and
                           (CurrentState = ST_ENABLE))
          else
            HADDR when (NextState = ST_READ)
```

```
        else
          HAddrReg;

-------------------------------------------------------------------------------
-- APB address decoding for slave devices
-- Decodes the address from HAddrMux, which only changes during a read or write
-- cycle.
-- When an address is used that is not in any of the ranges specified,
-- operation of the system continues, but no PSEL lines are set, so no
-- peripherals are selected during the read/write transfer.
-- Operation of PWDATA, PWRITE, PENABLE and PADDR continues as normal.
-------------------------------------------------------------------------------
p_AddrDecodeComb : process (HAddrMux, HRESETn)
begin
  if (HRESETn = '0') then
      PSELREGSInt   <= '0';
      PSELINTCInt   <= '0';
  elsif (HAddrMux(27 downto 24) = REGSBASE) then
      PSELREGSInt   <= '1';
      PSELINTCInt   <= '0';
  elsif (HAddrMux(27 downto 24) = INTCBASE) then
      PSELINTCInt   <= '1';
      PSELREGSInt   <= '0';
  else
      PSELREGSInt   <= '0';
      PSELINTCInt   <= '0';
  end if;
end process p_AddrDecodeComb;

WrBurst <= '1' when ((Valid = '1') and (RegValid = '1') and (HWRITE = '1'))
       else
          '0';

-------------------------------------------------------------------------------
-- Write Burst detection
-- The operation of the bridge is different when doing a single write operation
-- to a burst of writes, so need to be able to tell when a burst of writes is
-- being performed.
-- It is stored in a register to allow it to still be valid when the AHB starts
-- on the next transfer at the end of a burst.
-------------------------------------------------------------------------------
p_WrBurstSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    WrBurstReg <= '0';
  elsif (HCLK'event and HCLK = '1') then
    if HREADYIn = '1' then
      WrBurstReg <= WrBurst;
    end if;
  end if;
end process p_WrBurstSeq;


Wr2Rd <= '1' when ((CurrentState = ST_WRITE) and (HTransReg = TRN_SEQ) and
                   (HWriteReg = '0'))
    else
       '0';

-------------------------------------------------------------------------------
-- Write to read transfer
-- When a sequential transfer changes from a write to a read then an extra wait
-- state must be added due to the extra cycle taken to start a write transfer,
-- to allow time for the read data to be generated before being sampled on the
-- AHB.
-------------------------------------------------------------------------------
p_Wr2RdSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    Wr2RdReg <= '0';
  elsif (HCLK'event and HCLK = '1') then
    Wr2RdReg <= Wr2Rd;
  end if;
end process p_Wr2RdSeq;

-------------------------------------------------------------------------------
-- Next state logic for APB state machine
-- Generates next state from CurrentState and AHB inputs.

-- A normal non-seq transfer occurs on the AHB when the APB is not performing
-- any transfers.
-- A seq transfer occurs when the previous transfer on the AHB was also an APB
-- transfer.
-- A fast non-seq transfer is where there is one non-APB unwaited AHB cycle
-- between two APB transfers, so that the APB has just started the previous
-- transfer (current state is ST_READ or ST_WRITE) when the next transfer
-- appears on the AHB.
-- A slow non-seq is where a new APB transfer is started on the AHB when the
-- current state is ST_ENABLE, following a non-APB transfer.
-------------------------------------------------------------------------------
p_NextStateComb : process (CurrentState, Valid, HWRITE, WrBurstReg, Wr2RdReg,
                           RegValid, RegValidPrev, HWriteReg)
begin
```

```
case CurrentState is

  -- Idle state
  when ST_IDLE =>
    if (Valid = '1') then
      -- Non-seq, so enter write wait state
      if HWRITE = '1' then
        NextState <= ST_WWAIT;

      -- Non-seq, so straight to read state
      else
        NextState <= ST_READ;
      end if;
    else
      NextState   <= ST_IDLE;
    end if;

  -- Start of APB read cycle
  when ST_READ =>
    NextState     <= ST_ENABLE;

  -- Hold for one cycle before write
  when ST_WWAIT =>
    NextState     <= ST_WRITE;

  -- Start of APB write cycle
  when ST_WRITE =>
    NextState     <= ST_ENABLE;

  -- Enable cycle to complete transaction
  when ST_ENABLE =>
    if WrBurstReg = '1' then
      NextState   <= ST_WRITE;
    elsif (RegValid = '1' and RegValidPrev = '0') then
      if HWriteReg = '1' then
        NextState <= ST_WRITE;
      else
        NextState <= ST_READ;
      end if;
    elsif Wr2RdReg = '1' then
      NextState   <= ST_READ;
    elsif Valid = '1' then
      if HWRITE = '1' then
        NextState <= ST_WWAIT;
      else
        NextState <= ST_READ;
      end if;
    else
      NextState   <= ST_IDLE;
    end if;

  -- Return to idle on FSM error
  when others =>
    NextState     <= ST_IDLE;

  end case;
end process p_NextStateComb;

-------------------------------------------------------------------------------
-- State machine
-- Changes state on rising edge of HCLK.
-------------------------------------------------------------------------------
p_CurrentStSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    CurrentState <= ST_IDLE;
  elsif (HCLK'event and HCLK = '1') then
    CurrentState <= NextState;
  end if;
end process p_CurrentStSeq;

-------------------------------------------------------------------------------
-- APB enable generation
-- ApbEn set when preforming an access on the APB, that is, when the current
-- state is ST_READ or ST_WRITE, and is used to enable the APB output registers.
-------------------------------------------------------------------------------
ApbEn <= '1' when ((NextState = ST_READ) or (NextState = ST_WRITE))
      else
         '0';

-------------------------------------------------------------------------------
-- HREADYOut generation
-------------------------------------------------------------------------------
HReadyNext <= '0' when (((NextState = ST_WRITE) and (Valid = '1') and
                         (HSELAHBAPB = '1')) or
                        ((NextState = ST_ENABLE) and (Valid = '1') and
                         (RegValid = '0') and (HWRITE = '0')) or
                        (Wr2Rd = '1') or (NextState = ST_READ))
           else
              '1';
```

```
-- A registered version of HREADYOut is used to improve output timing.
-- Wait states are inserted during:
-- ST_WRITE when a write follows a read or a write.
-- ST_ENABLE when a read follows an AHB access that was not to the APB.
-- ST_ENABLE when a read follows a write (Wr2Rd set).
-- ST_READ.
-------------------------------------------------------------------------------
p_iHREADYOutSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    iHREADYOut <= '0';
  elsif (HCLK'event and HCLK = '1') then
    iHREADYOut <= HReadyNext;
  end if;
end process p_iHREADYOutSeq;

-------------------------------------------------------------------------------
-- Registered HWDATA for writes (PWDATA)
-------------------------------------------------------------------------------
PWDATAEn <= '1' when (NextState = ST_WRITE)
          else
             '0';

-------------------------------------------------------------------------------
-- Write wait state allows a register to be used to hold PWDATA.
-- Register enabled when in ST_WRITE.
-------------------------------------------------------------------------------
p_PWDATASeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
     PWDATA  <= (others => '0');
  elsif (HCLK'event and HCLK = '1') then
    if (PWDATAEn = '1') then
      PWDATA <= HWDATA;
    end if;
  end if;
end process p_PWDATASeq;

-------------------------------------------------------------------------------
-- Internal PENABLE generation
-- This is set to be a direct copy of one of the FSM bits. This should be
--  changed if the state encoding of the FSM is altered from the default
-------------------------------------------------------------------------------
iPENABLE <= CurrentState(2);

-------------------------------------------------------------------------------
-- iPSEL generation
-- Synchronous reset used to clear the PSEL outputs at the end of a transfer.
-- This logic is used to reduce the combinational output path from HTRANS to
-- the PSELxx registers, but it is functionally equivalent to:
-------------------------------------------------------------------------------
PSelRes <= '1' when ((CurrentState = ST_ENABLE) and
                     (WrBurstReg = '0') and (Wr2RdReg = '0') and
                     not ((RegValid = '1') and (RegValidPrev = '0')) and
                     not ((Valid = '1') and (HWRITE = '0')))
       else
          '0';

-------------------------------------------------------------------------------
-- Drives PSEL outputs with internal signals or set LOW on reset:
-- Set outputs with internal values when in ST_READ or ST_WRITE states
-- Reset outputs on system reset or at end of transfer (PSelRes HIGH)
-- Hold outputs when in ST_ENABLE state (ApbEn LOW)
-------------------------------------------------------------------------------
p_PSELSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    iPSELREGS     <= '0';
    iPSELINTC     <= '0';
  elsif (HCLK'event and HCLK = '1') then
    if (ApbEn = '1') then
      iPSELREGS   <= PSELREGSInt;
      iPSELINTC   <= PSELINTCInt;
    elsif (PSelRes = '1') then
      iPSELREGS   <= '0';
      iPSELINTC   <= '0';
    else
      iPSELREGS   <= iPSELREGS;
      iPSELINTC   <= iPSELINTC;
    end if;
  end if;
end process p_PSELSeq;

-------------------------------------------------------------------------------
-- Registered HADDR for reads and writes (iPADDR)
-- HAddrMux is used at the generation time of the APB address is different for
-- reads and writes, so both the direct and registered HADDR input need to be
-- used.
-- HAddrMux is captured by an ApbEn enabled register to generate iPADDR.
-- Signal iPADDR is used so that only (PADDRWIDTH - 1) of PADDR is driven.
-- iPADDR driven on State Machine change to ST_READ or ST_WRITE, with reset
-- to zero.
```

```
p_iPADDRSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    iPADDR <= (others => '0');
  elsif (HCLK'event and HCLK = '1') then
    if (ApbEn = '1') then
      iPADDR <= HAddrMux(PADDRWIDTH - 1 downto 0);
    end if;
  end if;
end process p_iPADDRSeq;

-- iPWRITE generation

NextPWRITE <= '1' when (NextState = ST_WRITE)
              else
                '0';

-- NextPWRITE is captured by an ApbEn enabled register to generate iPWRITE, and
-- is generated from NextState (set HIGH during a write cycle).
-- PWRITE output only changes when APB is accessed.

p_iPWRITESeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    iPWRITE    <= '0';
  elsif (HCLK'event and HCLK = '1') then
    if (ApbEn = '1') then
      iPWRITE <= NextPWRITE;
    end if;
  end if;
end process p_iPWRITESeq;

-- APB output drivers
-- Drive outputs with internal signals.

PADDR(PADDRWIDTH - 1 downto 0) <= iPADDR;

PWRITE    <= iPWRITE;

PENABLE   <= iPENABLE;

PSELREGS <= iPSELREGS;

PSELINTC <= iPSELINTC;

-- AHB output drivers
-- PRDATA is only ever driven during a read, so it can be directly copied to
-- HRDATA to reduce the output data delay onto the AHB.

HRDATA    <= PRDATA;

-- Drives the output port with the internal version, and sets it LOW at all
--  other times when the module is not selected.

HREADYOut <= iHREADYOut;

-- The response will always be OKAY to show that the transfer has been performed
-- successfully.

HRESP     <= RSP_OKAY;

end synth;

------------------------------------End
```

## Listing C.7: AHB to AHB Top Level

```
------------------------------------------------------------------------
-- This confidential and proprietary software may be used only as
-- authorised by a licensing agreement from ARM Limited
--   (C) COPYRIGHT 2000 ARM Limited
--       ALL RIGHTS RESERVED
-- The entire notice above must be reproduced on all authorised
-- copies and copies may only be made to the extent permitted
-- by a licensing agreement from ARM Limited.
--
------------------------------------------------------------------------
-- Version and Release Control Information:
--
-- File Name              : $RCSfile: AHBAHBTop.vhd,v $
-- File Revision          : $Revision: 1.1 $
--
-- Release Information    : $State: Exp $
```

```
--
-------------------------------------------------------------------------------
-- Purpose :
--           Example VHDL for Integrator Logic modules
--
-------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

-- library lmc;

-------------------------------------------------------------------------------

entity AHBAHBTop is
  port (
-- Inputs
        -- AHB interface
        HCLK            : in    std_logic; -- system bus clock
        TCK             : in    std_logic; -- test clock
        HRESETn         : in    std_logic; -- reset input (active low)
        HTRANS          : in    std_logic_vector(1 downto 0);
                                           -- AHB transfer type CONT [1:0]
        HWRITE          : in    std_logic; -- AHB hwrite CONT 11
        HSIZE           : in    std_logic_vector(1 downto 0);
                                           -- AHB hsize CONT [3:2]
        nPBUTT          : in    std_logic; -- push button used as
                                           -- interrupt source
        SW              : in    std_logic_vector(7 downto 0);
                                           -- switches
        HDRID           : in    std_logic_vector(3 downto 0);
                                           -- LM ID
        TDI             : in    std_logic; -- test data in
        HADDR           : in    std_logic_vector(31 downto 0);
                                           -- AHB address bus
-- Inouts
        HREADY          : inout std_logic; -- AHB ready CONT 12
        HDATA           : inout std_logic_vector(31 downto 0);
                                           -- AHB data bus (bi directional)
        SDATA           : inout std_logic_vector(31 downto 0);
                                           -- ZBT data bus
-- Outputs
        HRESP           : out   std_logic_vector(1 downto 0);
                                           -- AHB response CONT [14:13]
        HBUSREQ         : out   std_logic; -- AHB request
        HLOCK           : out   std_logic; -- AHB lock
        RTCK            : out   std_logic; -- return test clock
                                           -- (Multi-ICE feature)
        -- APB I/O (SLAVE 1)
        nLMINT          : out   std_logic; -- LM peripheral interrupt request
        LED             : out   std_logic_vector(8 downto 0);
                                           -- LED control
        CTRLCLK1        : out   std_logic_vector(18 downto 0);
                                           -- sets frequency of CLK1
        CTRLCLK2        : out   std_logic_vector(18 downto 0);
                                           -- sets frequency of CLK2
        -- ZBT RAM (SLAVE 2)
        SCLK            : out   std_logic; -- ZBT CLK
        SnWBYTE         : out   std_logic_vector(3 downto 0);
                                           -- ZBT byte write control signals
        SnOE            : out   std_logic; -- ZBT output enable
        SnCE            : out   std_logic; -- ZBT chip select
        SADVnLD         : out   std_logic; -- ZBT Mode pin
        SnWR            : out   std_logic; -- ZBT advance signal
        SnCKE           : out   std_logic; -- ZBT Clock enable signal
        SMODE           : out   std_logic; -- ZBT mode signal
        SADDR           : out   std_logic_vector(19 downto 2);
                                           -- ZBT address bus
        -- FLASH control signals
        FnOE            : out   std_logic; -- FLASH output enable
        FnWE            : out   std_logic; -- FLASH write enable
        -- misc signals
        PWRDNCLK1       : out   std_logic; -- clock power-down control
        PWRDNCLK2       : out   std_logic; -- clock power-down control
        TDO             : out   std_logic  -- test data out
       );
end AHBAHBTop;

-------------------------------------------------------------------------------
--
--                                AHBAHBTop
--
-------------------------------------------------------------------------------
--
-- Overview
-- ========
-- Top level VHDL for LM example
-- Connects AHB slaves to the Integrator AHB busses
--
-- Implements ...
--  AHB Decoder
```

```
--  AHB Slave to master Mux
--  AHB SSRAM Controller
--  AHB APB bridge
--  APB Interrupt controller
--  APB Register peripheral
--
------------------------------------------------------------------------
-------------------------------ARCHITECTURE-----------------------------

architecture synth of AHBAHBTop is

------------------------------------------------------------------------
-- Component declarations
------------------------------------------------------------------------

component AHBDecoder
  port (
        HCLK              : in    std_logic;
        HRESETn           : in    std_logic;
        HTRANS            : in    std_logic_vector(1 downto 0);
        HREADYIn          : in    std_logic;
        HDRID             : in    std_logic_vector(3 downto 0);
        HADDR             : in    std_logic_vector(31 downto 0);

        HSELAHBAPB        : out   std_logic;
        HSELSSRAM         : out   std_logic;
        HSELLOGICMODULE   : out   std_logic;
        HSELDefault       : out   std_logic;
        HREADYOut         : out   std_logic;
        HRESP             : out   std_logic_vector(1 downto 0)
       );
end component;

component AHBMuxS2M
  port (
        HCLK              : in    std_logic;
        HRESETn           : in    std_logic;
        HSELAHBAPB        : in    std_logic;
        HSELSSRAM         : in    std_logic;
        HREADYAHBAPB      : in    std_logic;
        HREADYSSRAM       : in    std_logic;
        HREADYDefault     : in    std_logic;
        HRESPAHBAPB       : in    std_logic_vector(1 downto 0);
        HRESPSSRAM        : in    std_logic_vector(1 downto 0);
        HRESPDefault      : in    std_logic_vector(1 downto 0);
        HREADYIn          : in    std_logic;
        HRDATAAHBAPB      : in    std_logic_vector(31 downto 0);
        HRDATASSRAM       : in    std_logic_vector(31 downto 0);

        HREADYOut         : out   std_logic;
        HRESP             : out   std_logic_vector(1 downto 0);
        HRDATA            : out   std_logic_vector(31 downto 0)
       );
end component;

component AHBAPBSys
  port (
        HCLK              : in    std_logic;
        HRESETn           : in    std_logic;
        HTRANS            : in    std_logic_vector(1 downto 0);
        HWRITE            : in    std_logic;
        HSELAHBAPB        : in    std_logic;
        HREADYIn          : in    std_logic;
        nPBUTT            : in    std_logic;
        SW                : in    std_logic_vector(7 downto 0);
        HWDATA            : in    std_logic_vector(31 downto 0);
        HADDR             : in    std_logic_vector(31 downto 0);

        HREADYOut         : out   std_logic;
        HRESP             : out   std_logic_vector(1 downto 0);
        nLMINT            : out   std_logic;
        CTRLCLK1          : out   std_logic_vector(18 downto 0);
        CTRLCLK2          : out   std_logic_vector(18 downto 0);
        LED               : out   std_logic_vector(8 downto 0);
        HRDATA            : out   std_logic_vector(31 downto 0)
       );
end component;

component AHBZBTRAM
  port (
        HCLK              : in    std_logic;
        HRESETn           : in    std_logic;
        HSELSSRAM         : in    std_logic;
        HREADYIn          : in    std_logic;
        HTRANS            : in    std_logic_vector(1 downto 0);
        HSIZE             : in    std_logic_vector(1 downto 0);
        HWRITE            : in    std_logic;
        HWDATA            : in    std_logic_vector(31 downto 0);
        SRDATA            : in    std_logic_vector(31 downto 0);
        HADDR             : in    std_logic_vector(31 downto 0);

        SCLK              : out   std_logic;
```

```
        HREADYOut       : out   std_logic;
        HRESP           : out   std_logic_vector(1 downto 0);
        SDATAEN         : out   std_logic;
        SnWBYTE         : out   std_logic_vector(3 downto 0);
        SnOE            : out   std_logic;
        SnCE            : out   std_logic;
        SADVnLD         : out   std_logic;
        SnWR            : out   std_logic;
        SMODE           : out   std_logic;
        SnCKE           : out   std_logic;
        SWDATA          : out   std_logic_vector(31 downto 0);
        HRDATA          : out   std_logic_vector(31 downto 0);
        SADDR           : out   std_logic_vector(19 downto 2)
        );
end component;

-- ----------------------------------------------------------------------
-- Constant declarations
-- ----------------------------------------------------------------------

-- ----------------------------------------------------------------------
-- Signal declarations
-- ----------------------------------------------------------------------

signal HSELAHBAPB       : std_logic;
-- peripheral select - APB Peripherals

signal HSELSSRAM        : std_logic;
-- peripheral select - SSRAM controller

signal HSELDefault      : std_logic;
-- peripheral select - default slave

signal HSELLOGICMODULE  : std_logic;
-- LM being addressed - used for response enable

signal HRDataApb        : std_logic_vector(31 downto 0);
-- read data bus from APB bridge

signal HRDATASSRAM      : std_logic_vector(31 downto 0);
-- read data bus from SSRAM

signal HRDATA           : std_logic_vector(31 downto 0);
-- read data bus from mux to master(s)

signal HReadyOutApb     : std_logic;
-- hready response from APB bridge

signal HReadyOutSsram   : std_logic;
-- hready response from SSRAM

signal HReadyOutDefault : std_logic;
-- hready response from default slave

signal HRespApb         : std_logic_vector(1 downto 0);
-- hresp response from APB

signal HRESPSSRAM       : std_logic_vector(1 downto 0);
-- hresp response from SSRAM

signal HRESPDefault     : std_logic_vector(1 downto 0);
-- hresp response from default slave

signal SWDATA           : std_logic_vector(31 downto 0);
-- SSRAM write data bus

signal SDATAEN          : std_logic;
-- tri-state enable for SSRAM data bus

signal iHReadyOut       : std_logic;
-- internal hready signal - feeds HREADYIn on slaves

signal ReadEnable       : std_logic;
-- used to control tri-states on top level

signal RespEnable       : std_logic;
-- used to control tri-states on top level

signal iHRespOut        : std_logic_vector(1 downto 0);
-- internal hresp from slave-master mux

-- ----------------------------------------------------------------------
-- Function declarations
-- ----------------------------------------------------------------------

-- ----------------------------------------------------------------------
--
-- Main body of code
-- =================
--
-- ----------------------------------------------------------------------

begin
```

```
-- Instantiation of AHBDecoder

uAHBDecoder : AHBDecoder
  port map (
            HCLK              => HCLK,
            HRESETn           => HRESETn,
            HTRANS            => HTRANS,
            HREADYIn          => HREADY,
            HDRID             => HDRID,
            HADDR             => HADDR,
            HSELAHBAPB        => HSELAHBAPB,
            HSELSSRAM         => HSELSSRAM,
            HSELLOGICMODULE   => HSELLOGICMODULE,
            HSELDefault       => HSELDefault,
            HREADYOut         => HReadyOutDefault,
            HRESP             => HRESPDefault
           );

-- Instantiation of AHBMuxS2M

uAHBMuxS2M : AHBMuxS2M
  port map (
            HCLK              => HCLK,
            HRESETn           => HRESETn,
            HSELAHBAPB        => HSELAHBAPB,
            HSELSSRAM         => HSELSSRAM,
            HREADYAHBAPB      => HReadyOutApb,
            HREADYSSRAM       => HReadyOutSsram,
            HREADYDefault     => HReadyOutDefault,
            HRESPAHBAPB       => HRespApb,
            HRESPSSRAM        => HRESPSSRAM,
            HRESPDefault      => HRESPDefault,
            HREADYIn          => HREADY,
            HRDATAAHBAPB      => HRDataApb,
            HRDATASSRAM       => HRDATASSRAM,
            HREADYOut         => iHReadyOut,
            HRESP             => iHRespOut,
            HRDATA            => HRDATA
           );

-- Instantiation of AHBAPBSys

uAHBAPBSys : AHBAPBSys
  port map (
            HCLK              => HCLK,
            HRESETn           => HRESETn,
            HTRANS            => HTRANS,
            HWRITE            => HWRITE,
            HSELAHBAPB        => HSELAHBAPB,
            HREADYIn          => HREADY,
            nPBUTT            => nPBUTT,
            SW                => SW,
            HWDATA            => HDATA,
            HADDR             => HADDR,
            HREADYOut         => HReadyOutApb,
            HRESP             => HRespApb,
            nLMINT            => nLMINT,
            CTRLCLK1          => CTRLCLK1,
            CTRLCLK2          => CTRLCLK2,
            LED               => LED,
            HRDATA            => HRDataApb
           );

-- Instantiation of AHBZBTRAM

uAHBZBTRAM : AHBZBTRAM
  port map (
            HCLK              => HCLK,
            HRESETn           => HRESETn,
            HSELSSRAM         => HSELSSRAM,
            HREADYIn          => HREADY,
            HTRANS            => HTRANS,
            HSIZE             => HSIZE,
            HWRITE            => HWRITE,
            HWDATA            => HDATA,
            SRDATA            => SDATA,
            HADDR             => HADDR,
            SCLK              => SCLK,
            HREADYOut         => HReadyOutSsram,
            HRESP             => HRESPSSRAM,
            SDATAEN           => SDATAEN,
            SnWBYTE           => SnWBYTE,
            SnOE              => SnOE,
            SnCE              => SnCE,
            SADVnLD           => SADVnLD,
            SnWR              => SnWR,
            SMODE             => SMODE,
            SnCKE             => SnCKE,
```

```
            SWDATA          => SWDATA,
            HRDATA          => HRDATASSRAM,
            SADDR           => SADDR
            );

-------------------------------------------------------------------------
-- Integrator System Bus uses tri-state muxing of data and slave response
-- signal respenable drives HRESP & HREADY out from this FPGA
-- readenable HRDATA, HRESP and HREADY
-------------------------------------------------------------------------
p_ReadEnSeq : process (HCLK, HRESETn)
begin
  if (HRESETn = '0') then
    ReadEnable <='0';
    RespEnable <='0';
  elsif (HCLK'event and HCLK = '1') then
    if (HREADY = '1') then
      -- start new data phase when HREADY ='1'
      if (HSELLOGICMODULE = '1') then
        RespEnable <='1';
        ReadEnable <= not (HWRITE);
      else
        ReadEnable <= '0';
        RespEnable <= '0';
      end if;
    end if;
  end if;
end process p_ReadEnSeq;

HREADY <= iHReadyOut when (RespEnable = '1')
        else
           'Z';

HRESP  <= iHRespOut when (RespEnable = '1')
        else
           (others => 'Z');

HDATA  <= HRDATA when (ReadEnable = '1')
        else
           (others => 'Z');

SDATA  <= SWDATA when (SDATAEN = '1')
        else
           (others => 'Z');

-- Ensure flash is disabled
FnOE   <= '1';
FnWE   <= '1';

-- Route virtual JTAG signals through. If this module is stacked with something
-- that uses a TAP controller, then it will still work OK
TDO  <= TDI;
RTCK <= TCK;

-- clk 1 running
PWRDNCLK1 <= '0';

-- clk 2 disabled as this is the SYSCLK OUT from the LM, must be disabled
-- when the LM is connected to a motherboard that supplies these clocks.
PWRDNCLK2 <= '1';

-- there is no master in this FPGA to ever request the bus
HBUSREQ   <= '0';

-- there is no master in this FPGA to ever request locked cycles
HLOCK     <= '0';

end synth;

------------------------------End--------------------------------
```

## Listing C.8: AHB to APB System

```
------------------------------------------------------------------------
-- This confidential and proprietary software may be used only as
-- authorised by a licensing agreement from ARM Limited
--   (C) COPYRIGHT 2000 ARM Limited
--       ALL RIGHTS RESERVED
-- The entire notice above must be reproduced on all authorised
-- copies and copies may only be made to the extent permitted
-- by a licensing agreement from ARM Limited.
--
------------------------------------------------------------------------
-- Version and Release Control Information:
--
-- File Name              : $RCSfile: AHBAPBSys.vhd,v $
-- File Revision          : $Revision: 1.1 $
--
-- Release Information    : $State: Exp $
--
```

```
-- Purpose :
--          Connects APB peripheral to the AHB-APB bridge
--
------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

------------------------------------------------------------------------

entity AHBAPBSys is
  port (
-- Inputs
        HCLK            : in    std_logic; -- system bus clock
        HRESETn         : in    std_logic; -- reset input (active low)
        HTRANS          : in    std_logic_vector(1 downto 0);
                                -- AHB transfer type
        HWRITE          : in    std_logic; -- AHB hwrite
        HSELAHBAPB      : in    std_logic; -- AHB peripheral select
        HREADYIn        : in    std_logic; -- AHB ready input
        nPBUTT          : in    std_logic; -- input that will be latched for
                                -- an interrupt example
        SW              : in    std_logic_vector(7 downto 0);
                                -- switches
        HWDATA          : in    std_logic_vector(31 downto 0);
                                -- AHB write data bus
        HADDR           : in    std_logic_vector(31 downto 0);
                                -- AHB address bus
-- Outputs
        HREADYOut       : out   std_logic; -- AHB ready output to S->M mux
        HRESP           : out   std_logic_vector(1 downto 0);
                                -- AHB response
        nLMINT          : out   std_logic; -- LM peripheral interrupt request
        CTRLCLK1        : out   std_logic_vector(18 downto 0);
                                -- sets frequency of CLK1
        CTRLCLK2        : out   std_logic_vector(18 downto 0);
                                -- sets frequency of CLK2
        LED             : out   std_logic_vector(8 downto 0);
                                -- LED control
        HRDATA          : out   std_logic_vector(31 downto 0)
                                -- AHB read data bus
        );

end AHBAPBSys;

------------------------------------------------------------------------
--
--                              AHBAPBSys
--                              =========
--
------------------------------------------------------------------------
--
-- Overview
-- ========
-- APB system glues together APB peripheral and connects them to the AHB-APB
-- bridge.
--
------------------------------------------------------------------------

------------------------------ARCHITECTURE------------------------------

architecture synth of AHBAPBSys is

------------------------------------------------------------------------
-- Component declarations
------------------------------------------------------------------------

component AHB2APB
  port (
        HCLK            : in    std_logic;
        HRESETn         : in    std_logic;
        HTRANS          : in    std_logic_vector(1 downto 0);
        HWRITE          : in    std_logic;
        HSELAHBAPB      : in    std_logic;
        HREADYIn        : in    std_logic;
        PRDATA          : in    std_logic_vector(31 downto 0);
        HWDATA          : in    std_logic_vector(31 downto 0);
        HADDR           : in    std_logic_vector(31 downto 0);

        PENABLE         : out   std_logic;
        HREADYOut       : out   std_logic;
        HRESP           : out   std_logic_vector(1 downto 0);
        PWRITE          : out   std_logic;
        PSELREGS        : out   std_logic;
        PSELINTC        : out   std_logic;
        PWDATA          : out   std_logic_vector(31 downto 0);
        HRDATA          : out   std_logic_vector(31 downto 0);
        PADDR           : out   std_logic_vector(31 downto 0)
        );
end component;

-- Registers peripheral
```

```
component APBRegs
  port (
        PCLK              : in    std_logic;
        nRESET            : in    std_logic;
        PENABLE           : in    std_logic;
        PSEL              : in    std_logic;
        PWRITE            : in    std_logic;
        nPBUTT            : in    std_logic;
        SW                : in    std_logic_vector(7 downto 0);
        PWDATA            : in    std_logic_vector(31 downto 0);
        PA                : in    std_logic_vector(6 downto 2);

        CTRLCLK1          : out   std_logic_vector(18 downto 0);
        CTRLCLK2          : out   std_logic_vector(18 downto 0);
        REGSINT           : out   std_logic;
        LED               : out   std_logic_vector(8 downto 0);
        PRDATA            : out   std_logic_vector(31 downto 0)
       );
end component;

-- Interrupt controller peripheral
component APBIntcon
  port (
        PCLK              : in    std_logic;
        nRESET            : in    std_logic;
        PENABLE           : in    std_logic;
        PSEL              : in    std_logic;
        PWRITE            : in    std_logic;
        INTSRC            : in    std_logic_vector(7 downto 4);
        PWDATA            : in    std_logic_vector(7 downto 0);
        PADDR             : in    std_logic_vector(7 downto 2);

        nLMINT            : out   std_logic;
        PRDATA            : out   std_logic_vector(7 downto 0)
       );
end component;

-------------------------------------------------------------------------
-- Constant declarations
-------------------------------------------------------------------------

-- used in muxing the APB read busses
constant ALLZERO32        : std_logic_vector(31 downto 0)
                          := "00000000000000000000000000000000";

-------------------------------------------------------------------------
-- Signal declarations
-------------------------------------------------------------------------

signal PA                 : std_logic_vector(31 downto 0);
-- Peripheral address bus

signal PWRITE             : std_logic;
-- Peripheral bus Write

signal PENABLE            : std_logic;
-- Peripheral Enable Signal

signal PWDATA             : std_logic_vector(31 downto 0);
-- APB write data

signal PRDATA             : std_logic_vector(31 downto 0);
-- APB read data

signal PSELREGS           : std_logic;
-- Peripheral select - register peripheral

signal PSELINTC           : std_logic;
-- Peripheral select - interrupt controller

signal INTSRC             : std_logic_vector(7 downto 4);
-- vector used to concatenate interrupt sources

signal RegPRData          : std_logic_vector(31 downto 0);
-- read data from register peripheral

signal RegPRDataMux       : std_logic_vector(31 downto 0);
-- read data presented to mux

signal IntcntPRData       : std_logic_vector(7 downto 0);
-- read data from interrupt controller

signal IntcntPRDataMux    : std_logic_vector(7 downto 0);
-- read data presented to mux

signal REGSINT            : std_logic;
-- interrupt from register peripheral

-------------------------------------------------------------------------
-- Function declarations
-------------------------------------------------------------------------

-------------------------------------------------------------------------
--
```

```
-- Main body of code
--==================
--

begin

-- Instantiation of AHB2APB

uAHB2APB : AHB2APB
  port map (
            HCLK          => HCLK,
            HRESETn       => HRESETn,
            HTRANS        => HTRANS,
            HWRITE        => HWRITE,
            HSELAHBAPB    => HSELAHBAPB,
            HREADYIn      => HREADYIn,
            PRDATA        => PRDATA,
            HWDATA        => HWDATA,
            HADDR         => HADDR,
            PENABLE       => PENABLE,
            HREADYOut     => HREADYOut,
            HRESP         => HRESP,
            PWRITE        => PWRITE,
            PSELREGS      => PSELREGS,
            PSELINTC      => PSELINTC,
            PWDATA        => PWDATA,
            HRDATA        => HRDATA,
            PADDR         => PA
           );

-- Instantiation of APBRegs

uAPBRegs : APBRegs
  port map (
            PCLK          => HCLK,
            nRESET        => HRESETn,
            PENABLE       => PENABLE,
            PSEL          => PSELREGS,
            PWRITE        => PWRITE,
            nPBUTT        => nPBUTT,
            SW            => SW,
            PWDATA        => PWDATA(31 downto 0),
            PA            => PA(6 downto 2),
            CTRLCLK1      => CTRLCLK1,
            CTRLCLK2      => CTRLCLK2,
            REGSINT       => REGSINT,
            LED           => LED,
            PRDATA        => RegPRData
           );

-- Instantiation of APBIntcon

uAPBIntcon : APBIntcon
  port map (
            PCLK          => HCLK,
            nRESET        => HRESETn,
            PENABLE       => PENABLE,
            PSEL          => PSELINTC,
            PWRITE        => PWRITE,
            INTSRC        => INTSRC,
            PWDATA        => PWDATA(7 downto 0),
            PADDR         => PA(7 downto 2),
            nLMINT        => nLMINT,
            PRDATA        => IntcntPRData
           );

-- MUX the read busses

RegPRDataMux      <= RegPRData when (PSELREGS = '1')
                     else
                     (others => '0');

IntcntPRDataMux <= IntcntPRData when (PSELINTC = '1')
                     else
                     (others => '0');

PRDATA            <= (ALLZERO32(31 downto 8) & IntcntPRDataMux(7 downto 0)) or
                        (RegPRDataMux(31 downto 0));

-- Concatenate interrupt sources. This example only uses one interrupt

-- 7:4 interrupt sources, 3:0 are soft interrupts assigned inside controller
INTSRC <= ('0' & '0' & '0' & REGSINT);

end synth;
```

```
-- ============================================End=============================
```

## Listing C.9: AHB Address Decoder

```
-- ============================================================================
-- This confidential and proprietary software may be used only as
-- authorised by a licensing agreement from ARM Limited
--   (C) COPYRIGHT 2000 ARM Limited
--       ALL RIGHTS RESERVED
-- The entire notice above must be reproduced on all authorised
-- copies and copies may only be made to the extent permitted
-- by a licensing agreement from ARM Limited.
--
-- ----------------------------------------------------------------------------
-- Version and Release Control Information:
--
-- File Name            : $RCSfile: AHBDecoder.vhd,v $
-- File Revision        : $Revision: 1.1 $
--
-- Release Information  : $State: Exp $
--
-- ----------------------------------------------------------------------------
-- Purpose :
--           Gives HSELx module select outputs to the AHB system slaves
--
-- ============================================================================

library IEEE;
use IEEE.std_logic_1164.all;

-- ----------------------------------------------------------------------------

entity AHBDecoder is
  port (
-- Inputs
        HCLK            : in  std_logic; -- system bus clock
        HRESETn         : in  std_logic; -- reset input (active low)
        HTRANS          : in  std_logic_vector(1 downto 0);
                                         -- AHB transfer type
        HREADYin        : in  std_logic; -- AHB ready input
        HDRID           : in  std_logic_vector(3 downto 0);
                                         -- header ID (stack position)
        HADDR           : in  std_logic_vector(31 downto 0);
                                         -- AHB address bus
-- Outputs
        HSELAHBAPB      : out std_logic; -- peripheral select - APB
                                         -- Peripherals
        HSELSSRAM       : out std_logic; -- peripheral select - SSRAM
                                         -- controller
        HSELLOGICMODULE : out std_logic; -- LM being addressed - used for
                                         -- response enable in top level
        HSELDefault     : out std_logic; -- peripheral select - default slave
        HREADYOut       : out std_logic; -- AHB ready output to S->M mux
        HRESP           : out std_logic_vector(1 downto 0)
                                         -- AHB Response signal

       );
end AHBDecoder;

-- ----------------------------------------------------------------------------
--
--                                  AHBDecoder
--                                  ==========
--
-- ----------------------------------------------------------------------------
--
-- Overview
-- ========
-- Provides the HSELx module select outputs to the AHB system slaves, and
-- controls the read data multiplexor. This module is specific to a particular
-- implementation.
--
-- The decoder block also contains the default slave. The hready and the
-- hresp outputs of the decoder are used when the APB bridge or the SSRAM
-- controller are not being addressed.
--
-- ----------------------------------------------------------------------------

-- ============================================ARCHITECTURE====================

architecture synth of AHBDecoder is

-- ----------------------------------------------------------------------------
-- Component declarations
-- ----------------------------------------------------------------------------

-- ----------------------------------------------------------------------------
-- Constant declarations
-- ----------------------------------------------------------------------------
-- HTRANS transfer type signal encoding
constant TRN_IDLE           : std_logic_vector(1 downto 0) := "00";
```

```
constant TRN_BUSY          : std_logic_vector(1 downto 0) := "01";
constant TRN_NONSEQ        : std_logic_vector(1 downto 0) := "10";
constant TRN_SEQ           : std_logic_vector(1 downto 0) := "11";

-- HRESP transfer response signal encoding
constant RSP_OKAY          : std_logic_vector(1 downto 0) := "00";
constant RSP_ERROR         : std_logic_vector(1 downto 0) := "01";
constant RSP_RETRY         : std_logic_vector(1 downto 0) := "10";
constant RSP_SPLIT         : std_logic_vector(1 downto 0) := "11";

-------------------------------------------------------------------------------
-- Signal declarations
-------------------------------------------------------------------------------
signal iHSELAHBAPB      : std_logic;
-- Internal copy of HSELAHBAPB

signal iHSELSSRAM       : std_logic;
-- Internal copy of HSELSSRAM

signal iHSELLOGICMODULE : std_logic;
-- Internal copy of HSELLOGICMODULE

signal iHSELDefault     : std_logic;
-- Internal copy of HSELDefault

signal NextHRESP        : std_logic_vector(1 downto 0);
-- D-input of HRESP

signal NextHREADY       : std_logic;
-- D-input of HREADY

signal iHREADYOut       : std_logic;
-- Internal copy of HREADYOut

-------------------------------------------------------------------------------
-- Function declarations
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
--
-- Main body of code
-- =================
--
-------------------------------------------------------------------------------

begin

-------------------------------------------------------------------------------
-- AHB address decoding for slave selection and read data multiplexor control
-------------------------------------------------------------------------------
iHSELLOGICMODULE   <= '1' when ((((HDRID = "1110") and
                                  (HADDR(31 downto 28) = "1100")) or
                                 ((HDRID = "0111") and
                                  (HADDR(31 downto 28) = "1101")) or
                                 ((HDRID = "1011") and
                                  (HADDR(31 downto 28) = "1110")) or
                                 ((HDRID = "1101") and
                                  (HADDR(31 downto 28) = "1111"))) and
                                (HRESETn = '1'))
                      else
                         '0';

iHSELAHBAPB        <= '1' when ((iHSELLOGICMODULE = '1') and
                                (HADDR(27 downto 25) = "000"))
                      else
                         '0';

iHSELSSRAM         <= '1' when ((iHSELLOGICMODULE = '1') and
                                (HADDR(27 downto 20) = "00100000"))
                      else
                         '0';

iHSELDefault       <= '1' when ((iHSELLOGICMODULE = '1') and not
                                (HADDR(27 downto 25) = "000") and not
                                (HADDR(27 downto 20) = "00100000"))
                      else
                         '0';

-------------------------------------------------------------------------------
-- Default slave output drivers
-------------------------------------------------------------------------------
NextHRESP <= RSP_ERROR when (((HTRANS = TRN_NONSEQ) or (HTRANS = TRN_SEQ)) and
                             (iHSELDefault = '1'))
             else
                RSP_OKAY;

-------------------------------------------------------------------------------
-- When an undefined area of the memory map is accessed, or an invalid address
-- is driven onto the address bus, the default slave outputs are selected and
-- passed to the current bus master.
-- An OKAY response is generated for IDLE or BUSY transfers to undefined
-- locations, but a two cycle ERROR response is generated if a non-sequential
```

```
-- or sequential transfer is attempted.
--------------------------------------------------------------------------------
p_HRESPSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    HRESP    <= RSP_OKAY;
  elsif (HCLK'event and HCLK = '1') then
    if iHREADYOut = '1' then
      HRESP <= NextHRESP;
    end if;
  end if;
end process p_HRESPSeq;

-- For the two cycle error response, HREADY is set LOW during the first cycle
-- and HIGH during the second cycle.
NextHREADY <= '1' when (iHREADYOut = '0')
              else
                 '0' when ((iHSELDefault = '1') and
                           (HTRANS = TRN_NONSEQ or HTRANS = TRN_SEQ))
              else
                 '1';

--------------------------------------------------------------------------------
-- Sequential logic for HREADYOut generation
--------------------------------------------------------------------------------
p_HREADYSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    iHREADYOut <= '1';
  elsif (HCLK'event and HCLK = '1') then
    iHREADYOut <= NextHREADY;
  end if;
end process p_HREADYSeq;

--------------------------------------------------------------------------------
-- Assign internal copy of signals to output ports
--------------------------------------------------------------------------------
HSELLOGICMODULE     <= iHSELLOGICMODULE;
HSELAHBAPB          <= iHSELAHBAPB;
HSELSSRAM           <= iHSELSSRAM;
HSELDefault         <= iHSELDefault;
HREADYOut           <= iHREADYOut;

end synth;

--------------------------------End-------------------------------------------
```

## Listing C.10: AHB Mux

```
--------------------------------------------------------------------------------
-- This confidential and proprietary software may be used only as
-- authorised by a licensing agreement from ARM Limited
--   (C) COPYRIGHT 2000 ARM Limited
--       ALL RIGHTS RESERVED
-- The entire notice above must be reproduced on all authorised
-- copies and copies may only be made to the extent permitted
-- by a licensing agreement from ARM Limited.
--
--------------------------------------------------------------------------------
-- Version and Release Control Information:
--
-- File Name              : $RCSfile: AHBMuxS2M.vhd,v $
-- File Revision          : $Revision: 1.1 $
--
-- Release Information    : $State: Exp $
--
--------------------------------------------------------------------------------
-- Purpose :
--           Central multiplexor - signals from slaves to masters
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------

entity AHBMuxS2M is
  port (
-- Inputs
        HCLK              : in    std_logic; -- system bus clock
        HRESETn           : in    std_logic; -- reset input (active low)
        HSELAHBAPB        : in    std_logic; -- AHB peripheral select -
                                             -- APB bridge
        HSELSSRAM         : in    std_logic; -- AHB peripheral select - SSRAM
        HREADYAHBAPB      : in    std_logic; -- hready from APB
        HREADYSSRAM       : in    std_logic; -- hready from SSRAM
        HREADYDefault     : in    std_logic; -- hready from default slave
                                             -- (in decoder)
        HRESPAHBAPB       : in    std_logic_vector(1 downto 0);
```

```
                                        -- hresponse from APB
        HRESPSSRAM      : in    std_logic_vector(1 downto 0);
                                        -- hresponse from SSRAM
        HRESPDefault    : in    std_logic_vector(1 downto 0);
                                        -- hresponse from default slave
        HREADYIn        : in    std_logic; -- hready in
        HRDATAAHBAPB    : in    std_logic_vector(31 downto 0);
                                        -- read data bus from APB
        HRDATASSRAM     : in    std_logic_vector(31 downto 0);
                                        -- read data bus from SSRAM
-- Outputs
        HREADYOut       : out   std_logic; -- muxed hready out
        HRESP           : out   std_logic_vector(1 downto 0);
                                        -- muxed response out
        HRDATA          : out   std_logic_vector(31 downto 0)
                                        -- muxed read data bus out to
                                        -- master(s)

    );
end AHBMuxS2M;

-------------------------------------------------------------------------------
--
--                              AHBMuxS2M
--                              =========
--
-------------------------------------------------------------------------------
--
-- Overview
-- ========
-- Central multiplexor - signals from slaves to masters
--
-------------------------------------------------------------------------------

---=============================ARCHITECTURE=================================--

architecture synth of AHBMuxS2M is

-------------------------------------------------------------------------------
-- Component declarations
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
-- Constant declarations
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
-- Signal declarations
-------------------------------------------------------------------------------
signal SelAHBAPB        : std_logic;
-- Select signal

signal Selssram         : std_logic;
-- Selecct SSRAM

signal iHREADY          : std_logic;
-- Internal HREADY used as HSEL register enable

-------------------------------------------------------------------------------
-- Function declarations
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Main body of code
-- =================
--
-------------------------------------------------------------------------------

begin

-------------------------------------------------------------------------------
-- HSEL registers
-- Registered HSEL outputs are needed to control the slave output multiplexors,
-- as the multiplexors must be switched in the cycle after the HSEL signals
-- have been driven.
-------------------------------------------------------------------------------
p_HSELSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    SelAHBAPB    <= '0';
    Selssram     <= '0';
  elsif (HCLK'event and HCLK = '1') then
    if HREADYIn = '1' then
      SelAHBAPB <= HSELAHBAPB;
      Selssram  <= HSELSSRAM;
    end if;
  end if;
end process p_HSELSeq;

-------------------------------------------------------------------------------
-- multiplexors controlling read data and responses from slaves to masters.
-------------------------------------------------------------------------------
```

```
HRDATA    <= HRDATAAHBAPB when (SelAHBAPB = '1')
             else
               HRDATASSRAM when (Selssram = '1')
             else
               "0000"&"0000"&"0000"&"0000"&"0000"&"0000"&"0000"&"0000";

iHREADY   <= HREADYAHBAPB when (SelAHBAPB = '1')
             else
               HREADYSSRAM when (Selssram = '1')
             else
               HREADYDefault;

HREADYOut <= iHREADY;

HRESP     <= HRESPAHBAPB when (SelAHBAPB = '1')
             else
               HRESPSSRAM when (Selssram = '1')
             else
               HRESPDefault;

end synth;

--------------------------------End-------------------------------
```

## Listing C.11: APB Registers

```
--------------------------------------------------------------------------------
-- This confidential and proprietary software may be used only as
-- authorised by a licensing agreement from ARM Limited
--   (C) COPYRIGHT 2000 ARM Limited
--       ALL RIGHTS RESERVED
-- The entire notice above must be reproduced on all authorised
-- copies and copies may only be made to the extent permitted
-- by a licensing agreement from ARM Limited.
--
--------------------------------------------------------------------------------
-- Version and Release Control Information:
--
-- File Name              : $RCSfile: APBRegs.vhd,v $
-- File Revision          : $Revision: 1.1 $
--
-- Release Information    : $State: Exp $
--
--------------------------------------------------------------------------------
-- Purpose :
--           This APB peripheral contains registers
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

--------------------------------------------------------------------------------

entity APBRegs is
  port (
-- Inputs
        PCLK            : in    std_logic; -- APB clock
        nRESET          : in    std_logic; -- AMBA reset
        PENABLE         : in    std_logic; -- APB enable
        PSEL            : in    std_logic; -- APB select
        PWRITE          : in    std_logic; -- APB read/write
        nPBUTT          : in    std_logic; -- input that will be latched for
                                           -- an interrupt example
        SW              : in    std_logic_vector(7 downto 0);
                                           -- switches
        PWDATA          : in    std_logic_vector(31 downto 0);
                                           -- APB write data
        PA              : in    std_logic_vector(6 downto 2);
                                           -- APB address bus
-- Outputs
        CTRLCLK1        : out   std_logic_vector(18 downto 0);
                                           -- sets frequency of CLK1
        CTRLCLK2        : out   std_logic_vector(18 downto 0);
                                           -- sets frequency of CLK2
        REGSINT         : out   std_logic; -- interrupt output
        LED             : out   std_logic_vector(8 downto 0);
                                           -- LED control
        PRDATA          : out   std_logic_vector(31 downto 0)
                                           -- APB read data
      );
end APBRegs;

--------------------------------------------------------------------------------
--
--                                APBRegs
--                                =======
--
```

```
--
-- Overview
-- ========
-- This APB peripheral contains registers to...
-- • program & lock the two clock oscillators
-- • write to the general purpose LEDs
-- • clear push button interrupt
-- • read the general purpose switches
--
-- SJD • Implements the DSP-RAM system
--
-- Certain registers are protected by the LOCK register. You must write 0xA05F
-- to the lock register to enable the following registers to be modified:
--
--   LM_OSC1
--   LM_OSC2
--
-- Provides nLMINT to the top level & registers all interrupt sources
--
-------------------------------------------------------------------------------

-------------------------------ARCHITECTURE-----------------------------------

architecture synth of APBRegs is

-------------------------------------------------------------------------------
-- Component declarations
-------------------------------------------------------------------------------

  component dspramController is

    generic ( dw    : integer := 16;
              sbw   : integer := 48;
              obits : integer := 1;
              nsv   : integer := 4;
              sv0   : integer := 32;
              sv1   : integer := 16;
              sv2   : integer := 8;
              sv3   : integer := 1;
              sv4   : integer := 2;
              sv5   : integer := 4;
              sv6   : integer := 6;
              sv7   : integer := 12;
              isr   : boolean := false;
              sd    : integer := 2;
              ndb   : integer := 4;
              nPE   : integer := 16;
              nra   : integer := 2;
              nla   : integer := 1;
              nrb   : integer := 2;
              nlb   : integer := 1 );

    port ( clock       : in  std_logic;
           nReset      : in  std_logic;
           chipEnable  : in  std_logic;
           APBSelect   : in  std_logic;
           writeEnable : in  std_logic;
           writeData   : in  std_logic_vector( 31 downto 0 );
           address     : in  std_logic_vector( 6 downto 2 );
           readData    : out std_logic_vector( 31 downto 0 ) );

  end component dspramController;

-------------------------------------------------------------------------------
-- Constant declarations
-------------------------------------------------------------------------------

-- 1 MHz default clock values
constant OSC1_VECTOR      : std_logic_vector(18 downto 0)
                          := "1100111110000000100";

constant OSC2_VECTOR      : std_logic_vector(18 downto 0)
                          := "1100111110000000100";

-- Lock register key 0xA05F
constant LOCK_KEY         : std_logic_vector(15 downto 0)
                          := "1010000001011111";

-- Address decoding
constant LM_OSC1          : std_logic_vector(6 downto 2) := "10110";
-- read/write

constant LM_OSC2          : std_logic_vector(6 downto 2) := "10111";
-- read/write

constant LM_LOCK          : std_logic_vector(6 downto 2) := "11000";
-- read/write

constant LM_LEDS          : std_logic_vector(6 downto 2) := "11001";
-- read/write

constant LM_INT           : std_logic_vector(6 downto 2) := "11010";
-- read/write
```

```
constant LMSW              : std_logic_vector(6 downto 2) := "11011";
-- read only

-------------------------------------------------------------------------
-- Signal declarations
-------------------------------------------------------------------------
signal LmOscReg1           : std_logic_vector(18 downto 0);
-- Oscillator register1

signal LmOscReg2           : std_logic_vector(18 downto 0);
-- Oscillator register2

signal LmLckReg            : std_logic_vector(15 downto 0);
-- Lock register

signal LmLedsReg           : std_logic_vector(8 downto 0);
-- LED register

signal LmIntReg            : std_logic;
-- INT register

signal LmSwReg             : std_logic_vector(7 downto 0);
-- Switch register

signal Locked              : std_logic;
-- Registers are Locked

signal NextPRDATA          : std_logic_vector(31 downto 0);
-- read data

signal dspramReadData      : std_logic_vector( 31 downto 0 );

-------------------------------------------------------------------------
-- Function declarations
-------------------------------------------------------------------------


-------------------------------------------------------------------------
--
-- Main body of code
-- =================
--
-------------------------------------------------------------------------

begin

-- Locked signal protects registers that could be accidently changed
Locked <= '0' when (LmLckReg = LOCK_KEY)
          else
            '1';

-- switch register is read only
LmSwReg  <= SW;

-------------------------------------------------------------------------
-- Lock register is read/write
-------------------------------------------------------------------------
p_LdLckRegSeq : process(PCLK, nRESET)
begin
  if (nRESET = '0') then
    LmLckReg       <= (others => '0');
  elsif (PCLK'event and PCLK = '1') then
    if ((PSEL and PWRITE and PENABLE) = '1') then
      if (PA = LM_LOCK) then
        LmLckReg <= PWDATA(15 downto 0);
      end if;
    end if;
  end if;
end process p_LdLckRegSeq;

-------------------------------------------------------------------------
-- Oscillator1 register is read/write, protected by lock register
-------------------------------------------------------------------------
p_LdOscRegSeq1 : process(PCLK, nRESET)
begin
  if (nRESET = '0') then
    LmOscReg1      <= OSC1_VECTOR;
  elsif (PCLK'event and PCLK = '1') then
    if ((PSEL and PWRITE and PENABLE and not Locked) = '1') then
      if (PA = LM_OSC1) then
        LmOscReg1 <= PWDATA(18 downto 0);
      end if;
    end if;
  end if;
end process p_LdOscRegSeq1;

CTRLCLK1 <= LmOscReg1;

-------------------------------------------------------------------------
-- Oscillator2 register is read/write, protected by lock register
-------------------------------------------------------------------------
p_LdOscRegSeq2 : process(PCLK, nRESET)
```

```
begin
  if (nRESET = '0') then
    LmOscReg2       <= OSC2_VECTOR;
  elsif (PCLK'event and PCLK = '1') then
    if ((PSEL and PWRITE and PENABLE and not Locked) = '1') then
      if (PA = LM_OSC2) then
        LmOscReg2 <= PWDATA(18 downto 0);
      end if;
    end if;
  end if;
end process p_LdOscRegSeq2;

CTRLCLK2 <= LmOscReg2;

-------------------------------------------------------------------------
-- LEDS register is read/write
-------------------------------------------------------------------------

p_LdLEDSRegSeq : process(PCLK, nRESET)
begin
  if (nRESET = '0') then
  -- put a pattern on them
    LmLedsReg       <= "101010101";
  elsif (PCLK'event and PCLK = '1') then
    if ((PSEL and PWRITE and PENABLE) = '1') then
      if (PA = LM_LEDS) then
        LmLedsReg <= PWDATA(8 downto 0);
      end if;
    end if;
  end if;
end process p_LdLEDSRegSeq;

--LED <= LmLedsReg;

-------------------------------------------------------------------------
-- interrupt is latched on rising edge of nPBUTTution input
-- INT register is read/write(to clear int)
-------------------------------------------------------------------------

p_LdIntRegSeq : process(PCLK, nRESET, nPBUTT)
begin
  if (nRESET = '0') then
    LmIntReg        <= '0';
  elsif (nPBUTT = '0') then
    LmIntReg        <= '1';
  elsif (PCLK'event and PCLK = '1') then
    if ((PSEL and PWRITE and PENABLE) = '1') then
      if (PA = LM_INT) then
        LmIntReg <= PWDATA(0);
      end if;
    end if;
  end if;
end process p_LdIntRegSeq;

REGSINT <= LmIntReg;

-- Place the DSP-RAM controller in
dspram : dspramController generic map ( dw     => 16,
                                        sbw    => 16,
                                        obits  => 1,
                                        nsv    => 4,
                                        sv0    => 32,
                                        sv1    => 16,
                                        sv2    => 8,
                                        sv3    => 1,
                                        sv4    => 2,
                                        sv5    => 4,
                                        sv6    => 6,
                                        sv7    => 12,
                                        isr    => false,
                                        sd     => 2,
                                        ndb    => 5,
                                        nPE    => 32,
                                        nra    => 2,
                                        nia    => 1,
                                        nrb    => 2,
                                        nib    => 1 )

                         port map ( clock        => PCLK,
                                    nReset       => nRESET,
                                    chipEnable   => PENABLE,
                                    APBSelect    => PSEL,
                                    writeEnable  => PWRITE,
                                    writeData    => PWDATA,
                                    address      => PA,
                                    readData     => dspramReadData );

LED <= not dspramReadData( 8 downto 0 );

-------------------------------------------------------------------------
-- Read registers
-------------------------------------------------------------------------

p_GenNPRDATAComb : process (PA, LmOscReg1, LmOscReg2, LmLckReg, Locked,
                            LmLedsReg, LmIntReg, LmSwReg, dspramReadData )
```

```
begin
  NextPRDATA <= (others => '0');
  case PA is
    when LM_OSC1 =>
      NextPRDATA(18 downto 0) <= LmOscReg1;
    when LM_OSC2 =>
      NextPRDATA(18 downto 0) <= LmOscReg2;
    when LM_LOCK =>
      NextPRDATA(15 downto 0) <= LmLckReg;
      NextPRDATA(16)          <= Locked;
    when LM_LEDS =>
      NextPRDATA(8 downto 0)  <= LmLedsReg;
    when LM_INT =>
      NextPRDATA(0)           <= LmIntReg;
    when LM_SW =>
      NextPRDATA(7 downto 0)  <= LmSwReg;

    when "00000" |
         "00001" |
         "00010" |
         "00011" |
         "00100" |
         "00101" |
         "00110" |
         "00111" |
         "01000" |
         "01001" |
         "01010" |
         "01011" |
         "01100" |
         "01101" |
         "01110" |
         "01111" |
         "10000" |
         "10001" |
         "10010" |
         "10011" |
         "10100" |
         "10101" =>
      NextPRData <= dspramReadData;

    when others =>
      NextPRDATA(31 downto 0) <="00000000000000000000000000000000";
  end case;
end process p_GenNPRDATAComb;

-------------------------------------------------------------------------------
-- When the peripheral is not being accessed, '0's are driven
-- on the Read Databus (PRDATA) so as not to place any restrictions
-- on the method of external bus connection. The external data buses of the
-- peripherals on the APB may then be connected to the ASB-to-APB bridge using
-- Muxed or ORed bus connection method.
-------------------------------------------------------------------------------
p_RdSeq : process (PCLK, nRESET)
begin
  if (nRESET = '0') then
    PRDATA  <= (others => '0');
  elsif (PCLK'event and PCLK = '1') then
    PRDATA  <= NextPRDATA;
  end if;
end process p_RdSeq;

end synth;

------------------------------------End-------------------------------------
```

## Listing C.12: APB Interrupt Controller

```
--------------------------------------------------------------------------------
-- This confidential and proprietary software may be used only as
-- authorised by a licensing agreement from ARM Limited
--   (C) COPYRIGHT 2000 ARM Limited
--       ALL RIGHTS RESERVED
-- The entire notice above must be reproduced on all authorised
-- copies and copies may only be made to the extent permitted
-- by a licensing agreement from ARM Limited.
--
-------------------------------------------------------------------------------
-- Version and Release Control Information:
--
-- File Name              : $RCSfile: APBIntcon.vhd,v $
-- File Revision          : $Revision: 1.1 $
--
-- Release Information    : $State: Exp $
--
-------------------------------------------------------------------------------
-- Purpose :
--           Provides nLMINT to the top level & registers all interrupt sources
--
--------------------------------------------------------------------------------
```

```
library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------

entity APBIntcon is
  port (
-- Inputs
        PCLK            : in    std_logic; -- APB peripheral clock
        nRESET          : in    std_logic; -- reset input
        PENABLE         : in    std_logic; -- APB peripheral enable
        PSEL            : in    std_logic; -- APB peripheral select
        PWRITE          : in    std_logic; -- APB write
        INTSRC          : in    std_logic_vector(7 downto 4);
                                -- interrupt source vector
        PWDATA          : in    std_logic_vector(7 downto 0);
                                -- APB write data bus
        PADDR           : in    std_logic_vector(7 downto 2);
                                -- APB address bus
-- Outputs
        nLMINT          : out   std_logic; -- interrupt output
        PRDATA          : out   std_logic_vector(7 downto 0)
                                -- APB read data bus
        );
end APBIntcon;

--------------------------------------------------------------------------------
--
--                                  APBIntcon
--                                  =========
--
--------------------------------------------------------------------------------
--
-- Overview
-- ========
-- Provides nLMINT to the top level & registers, all interrupt sources
--
--------------------------------------------------------------------------------

---=============================ARCHITECTURE==================================--

architecture synth of APBIntcon is

--------------------------------------------------------------------------------
-- Component declarations
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
-- Constant declarations
--------------------------------------------------------------------------------
-- Number if interrupt sources
-- This example only uses 1 real & 4 soft interrupts, the AxBAPBSys block
-- assigns the top three as static zeros to show where to concatenate IRQ
-- sources
constant NUMLMINTS          : integer := 8;

--------------------------------------------------------------------------------
--  Address offset constants - see Reference Peripheral specification
--------------------------------------------------------------------------------
-- bits 4:2 register
-- bit 5 irq/fiq
-- bit 7:6 processor number
constant LM_ISTAT           : std_logic_vector(5 downto 2) := "0000";
constant LM_IRSTAT          : std_logic_vector(5 downto 2) := "0001";
constant LM_IENSET          : std_logic_vector(5 downto 2) := "0010";
constant LM_IENCLR          : std_logic_vector(5 downto 2) := "0011";
constant LM_SOFTINT         : std_logic_vector(5 downto 2) := "0100";

--------------------------------------------------------------------------------
-- Signal declarations
--------------------------------------------------------------------------------
signal RawIntReg        : std_logic_vector(NUMLMINTS-1 downto 0);
-- Raw Interrupt register

signal SoftIntReg       : std_logic_vector(3 downto 0);
-- Soft Interrupt register

signal LmIntEnReg       : std_logic_vector(NUMLMINTS-1 downto 0);
-- Interrupt Enable Register

signal LmIntStReg       : std_logic_vector(NUMLMINTS-1 downto 0);
-- Interrupt status register

signal DataBusEn        : std_logic;
-- Data bus Enable signal

signal NextPRDATA       : std_logic_vector(NUMLMINTS-1 downto 0);
-- D-input of PRDATA

signal inLMINT          : std_logic;
-- Internal copy of nLMINT
```

```
-- Function declarations

-- OrVectorIrq :
-- This function performs logical OR of all inputs
function OrVectorIrq (InVector : std_logic_vector(NUMLMINTS-1 downto 0))
    return std_logic is
variable Temp : std_logic;
-- Temporary variable

begin
  Temp := InVector(0);
  for i in NUMLMINTS-1 downto 1 loop
    Temp := Temp or InVector(i);
  end loop;
  return (Temp);
end OrVectorIrq;

--
-- Main body of code
--

begin

-- nLMINT output is the logical OR of all enabled interrupt sources
inLMINT <= OrVectorIrq(LmIntStReg);

-- synchronize nLMINT to the rising edge of the clock and
-- ensure interrupts are cleared during reset
p_SyncnLMINT : process(PCLK, nRESET)
begin
  if (nRESET = '0') then
    nLMINT <= '1';
  elsif (PCLK'event and PCLK = '1') then
    nLMINT <= not inLMINT;
  end if;
end process p_SyncnLMINT;

-- Status registers are the logical AND of the pending interrupts
-- and the appropriate enable register
LmIntStReg <= RawIntReg and LmIntEnReg;

-- Raw interrupts are registered on the rising edge of the clock
p_SyncRawIntSeq : process (PCLK, nRESET)
begin
  if (nRESET = '0') then
    RawIntReg(NUMLMINTS-1 downto 4) <= (others => '0');
    RawIntReg(3 downto 0)           <= "0000";
  elsif (PCLK'event and PCLK = '1') then
    RawIntReg(NUMLMINTS-1 downto 4) <= INTSRC(NUMLMINTS-1 downto 4);
    RawIntReg(3 downto 0)           <= SoftIntReg;
  end if;
end process p_SyncRawIntSeq;

-- Enable registers have seperate set and clear operations
p_WrEnRegSeq : process(PCLK, nRESET)
begin
  if (nRESET = '0') then
    LmIntEnReg <= (others => '0');
  elsif (PCLK'event and PCLK = '1') then
    if ((PSEL and PWRITE and PENABLE) = '1') then

      -- set IRQ enable bits
      if (PADDR(5 downto 2) = LM_IENSET) then
        LmIntEnReg <= PWDATA(NUMLMINTS-1 downto 0) or LmIntEnReg;
      end if;

      -- clear IRQ enable bits
      if (PADDR(5 downto 2) = LM_IENCLR) then
        LmIntEnReg <= (not PWDATA(NUMLMINTS-1 downto 0)) and LmIntEnReg;
      end if;

    end if;
  end if;
end process p_WrEnRegSeq;

-- Soft interrupt register
```

```
p_WrSoftRegSeq : process (PCLK, nRESET)
begin
  if (nRESET = '0') then
    SoftIntReg  <= "0000";
  elsif (PCLK'event and PCLK = '1') then

    if (((PSEL and PWRITE and PENABLE) = '1') and
        (PADDR(5 downto 2) = LM_SOFTINT)) then
      SoftIntReg <= PWDATA(3 downto 0);
    end if;
  end if;
end process p_WrSoftRegSeq;

-- drive PD when registers selected for read
DataBusEn <= PSEL and (not PWRITE);

-- multiplexor to read registers depending on address
NextPRDATA <= RawIntReg when ((PADDR(5 downto 2) = LM_IRSTAT) and
                              (DataBusEn = '1'))
          else
             LmIntStReg when ((PADDR(5 downto 2) = LM_ISTAT) and
                              (DataBusEn = '1'))
          else
             LmIntEnReg when ((PADDR(5 downto 2) = LM_IENSET) and
                              (DataBusEn = '1'))
          else
             (others => '0');
```

```
-- When the peripheral is not being accessed, '0's are driven
-- on the Read Databus (PRDATA) so as not to place any restrictions
-- on the method of external bus connection. The external data buses of the
-- peripherals on the APB may then be connected to the ASB-to-APB bridge using
-- Muxed or ORed bus connection method.
```

```
p_RdRegSeq : process (PCLK, nRESET)
begin
  if (nRESET = '0') then
    PRDATA  <= (others => '0');
  elsif (PCLK'event and PCLK = '1') then
    PRDATA  <= NextPRDATA;
  end if;
end process p_RdRegSeq;

end synth;
```

```
------------------------------------End-------------------------------------
```

## Listing C.13: AHB SRAM Interface

```
------------------------------------------------------------------------------
-- This confidential and proprietary software may be used only as
-- authorised by a licensing agreement from ARM Limited
--   (C) COPYRIGHT 2000 ARM Limited
--       ALL RIGHTS RESERVED
-- The entire notice above must be reproduced on all authorised
-- copies and copies may only be made to the extent permitted
-- by a licensing agreement from ARM Limited.
--
------------------------------------------------------------------------------
-- Version and Release Control Information:
--
-- File Name              : $RCSfile: AHBZBTRAM.vhd,v $
-- File Revision          : $Revision: 1.1 $
--
-- Release Information    : $State: Exp $
--
------------------------------------------------------------------------------
-- Purpose :
--           AHB ZBT SRAM controller
--
------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

------------------------------------------------------------------------------

entity AHBZBTRAM is
  port (
-- Inputs
       HCLK            : in    std_logic; -- system bus clock
       HRESETn         : in    std_logic; -- reset input (active low)
       HSELSSRAM       : in    std_logic; -- AHB peripheral select
       HREADYin        : in    std_logic; -- AHB ready input
       HTRANS          : in    std_logic_vector(1 downto 0);
                                          -- AHB transfer type
       HSIZE           : in    std_logic_vector(1 downto 0);
                                          -- AHB hsize
```

```
        HWRITE          : in    std_logic; -- AHB hwrite
        HWDATA          : in    std_logic_vector(31 downto 0);
                                -- AHB write data bus
        SRDATA          : in    std_logic_vector(31 downto 0);
                                -- SSRAM read data bus
        HADDR           : in    std_logic_vector(31 downto 0);
                                -- AHB address bus
-- Outputs
        SCLK            : out   std_logic; -- SSRAM clock
        HREADYOut       : out   std_logic; -- AHB ready output to S->M mux
        HRESP           : out   std_logic_vector(1 downto 0);
                                -- AHB response
        SDATAEN         : out   std_logic; -- SSRAM tristate enable
        SnWBYTE         : out   std_logic_vector(3 downto 0);
                                -- SSRAM byte lane writes
        SnOE            : out   std_logic; -- SSRAM output enable
        SnCE            : out   std_logic; -- SSRAM chip enable
        SADVnLD         : out   std_logic; -- SSRAM advance / load
        SnWR            : out   std_logic; -- SSRAM write
        SMODE           : out   std_logic; -- SSRAM write MODE
        SnCKE           : out   std_logic; -- SSRAM write Clock enable
        SWDATA          : out   std_logic_vector(31 downto 0);
                                -- SSRAM write data bus
        HRDATA          : out   std_logic_vector(31 downto 0);
                                -- AHB read data bus
        SADDR           : out   std_logic_vector(19 downto 2)
                                -- SSRAM address bus
    );
end AHBZBTRAM;

--------------------------------------------------------------------------------
--
--                                  AHBZBTRAM
--                                  =========
--
--------------------------------------------------------------------------------
--
-- Overview
-- ========
-- AHB ZBT SRAM controller
--
--------------------------------------------------------------------------------

---=============================ARCHITECTURE=================================---

architecture synth of AHBZBTRAM is

--------------------------------------------------------------------------------
-- Component declarations
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
-- Constant declarations
--------------------------------------------------------------------------------
constant ST_IDLE            : std_logic_vector(1 downto 0) := "00";
constant ST_READ            : std_logic_vector(1 downto 0) := "01";
constant ST_WRITE           : std_logic_vector(1 downto 0) := "10";

-- HTRANS transfer type signal encoding
constant TRN_IDLE           : std_logic_vector(1 downto 0) := "00";
constant TRN_BUSY           : std_logic_vector(1 downto 0) := "01";
constant TRN_NONSEQ         : std_logic_vector(1 downto 0) := "10";
constant TRN_SEQ            : std_logic_vector(1 downto 0) := "11";

-- HRESP transfer response signal encoding
constant RSP_OKAY           : std_logic_vector(1 downto 0) := "00";
constant RSP_ERROR          : std_logic_vector(1 downto 0) := "01";
constant RSP_RETRY          : std_logic_vector(1 downto 0) := "10";
constant RSP_SPLIT          : std_logic_vector(1 downto 0) := "11";

--------------------------------------------------------------------------------
-- Signal declarations
--------------------------------------------------------------------------------
signal Valid                : std_logic;
-- Module is selected with valid transfer

signal NextState            : std_logic_vector(1 downto 0);
-- State machine

signal CurrentState         : std_logic_vector(1 downto 0);
-- Current state

--------------------------------------------------------------------------------
-- Function declarations
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
--
-- Main body of code
---==================
--
--------------------------------------------------------------------------------
```

```
begin

-------------------------------------------------------------------------
-- Valid transfer detection
-- The slave must only respond to a valid transfer, so this must be detected.
-- Valid AHB transfers only take place when a non-sequential or sequential
-- transfer is shown on HTRANS - an idle or busy transfer should be ignored.
-------------------------------------------------------------------------
Valid <= '1' when ((HSELSSRAM = '1') and (HREADYIn = '1') and
                   ((HTRANS = TRN_NONSEQ) or (HTRANS = TRN_SEQ)))
      else
        '0';

-------------------------------------------------------------------------
-- Next state logic for APB state machine
-- Generates next state from CurrentState and AHB inputs.
-------------------------------------------------------------------------
p_NextStateComb : process (CurrentState, Valid, HWRITE)
begin
  case CurrentState is

    -- Idle state
    when ST_IDLE =>
      if (Valid = '1') then
        if (HWRITE = '1') then
          NextState <= ST_WRITE;
        else
          NextState <= ST_READ;
        end if;
      else
        NextState   <= ST_IDLE;
      end if;

    -- second read cycle
    when ST_READ =>
      if (Valid = '1') then
        if (HWRITE = '1') then
          NextState <= ST_WRITE;
        else
          NextState <= ST_READ;
        end if;
      else
        NextState   <= ST_IDLE;
      end if;

    -- second write cycle
    when ST_WRITE =>
      if (Valid = '1') then
        if (HWRITE = '1') then
          NextState <= ST_WRITE;
        else
          NextState <= ST_READ;
        end if;
      else
        NextState   <= ST_IDLE;
      end if;

    -- Return to idle on FSM error
    when others =>
      NextState     <= ST_IDLE;

  end case;
end process p_NextStateComb;

-------------------------------------------------------------------------
-- Signals controlled by statemachine
-------------------------------------------------------------------------
-- SRAM DATA tristate control passed to top top level
SDATAEN   <= '1' when (CurrentState = ST_WRITE)
          else
            '0';

-- ensure that ZBT cannot drive out during a write
  SnOE    <= '1' when (CurrentState = ST_WRITE)
          else
            '0';

-------------------------------------------------------------------------
-- State machine
-- Changes state on rising edge of HCLK.
-------------------------------------------------------------------------
p_CurrentStSeq : process (HRESETn, HCLK)
begin
  if (HRESETn = '0') then
    CurrentState <= ST_IDLE;
  elsif (HCLK'event and HCLK = '1') then
    CurrentState <= NextState;
  end if;
end process p_CurrentStSeq;

-------------------------------------------------------------------------
```

```
-- ZBT SRAM output drivers
-------------------------------------------------------------------------
SWDATA      <=HWDATA;

SADDR       <=HADDR(19 downto 2);

SCLK        <=HCLK;

SnWBYTE(0)  <= '0' when (((HADDR(1 downto 0) = "00") and
                          (HSIZE(1 downto 0) = "00")) or
                         ((HADDR(1) = '0') and (HSIZE(1 downto 0) = "01")) or
                         ((HSIZE(1 downto 0) = "10")))
            else
               '1';

SnWBYTE(1)  <= '0' when (((HADDR(1 downto 0) = "01") and
                          (HSIZE(1 downto 0) = "00")) or
                         ((HADDR(1) = '0') and (HSIZE(1 downto 0) = "01")) or
                         ((HSIZE(1 downto 0) = "10")))
            else
               '1';

SnWBYTE(2) <= '0' when (((HADDR(1 downto 0) = "10") and
                          (HSIZE(1 downto 0) = "00")) or
                         ((HADDR(1) = '1') and (HSIZE(1 downto 0) = "01")) or
                         ((HSIZE(1 downto 0) = "10")))
            else
               '1';


SnWBYTE(3) <= '0' when (((HADDR(1 downto 0) = "11") and
                          (HSIZE(1 downto 0) = "00")) or
                         ((HADDR(1) = '1') and (HSIZE(1 downto 0) = "01")) or
                         ((HSIZE(1 downto 0) = "10")))
            else
               '1';

SnCE        <= '0' when (Valid ='1')
            else
               '1';

SnWR        <= not (HWRITE);

SADVnLD     <= '0';

SMODE       <= '0';

SnCKE       <= '0';

-------------------------------------------------------------------------
-- AHB output drivers
-------------------------------------------------------------------------
HRDATA      <= SRDATA;
HRESP       <= RSP_OKAY;
HREADYOut   <= '1';

end synth;

-------------------------------------End-------------------------------------
```

# C.2 DSP-RAM PE VHDL

## C.2.1 Logic

Listing C.14: Logic Package

```
--
-- logicPackage.vhd
--
-- Steve Dillen
-- Jun 13, 2002
--
-- $Id: logicPackage.vhd,v 1.5 2003/03/14 18:11:05 dillen Exp $
--
-- $Log: logicPackage.vhd,v $
-- Revision 1.5  2003/03/14 18:11:05  dillen
-- Removed muxN, demuxN, decoderN
--
-- Revision 1.4  2003/03/14 18:04:27  dillen
-- Added DFF to package
--
-- Revision 1.3  2002/08/16 16:31:08  dillen
-- Added bidirectional Driver
```

```
--
-- Revision 1.2  2002/07/18 19:06:13  dillen
-- Made the demux generic and added it to the logic package
--
-- Revision 1.1  2002/07/17 17:27:34  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

package logicPackage is

  component mux2 is

    generic ( busWidth : integer := 4 );

    port ( ip0 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip1 : in  std_logic_vector( busWidth - 1 downto 0 );
           sel : in  std_logic;
           op  : out std_logic_vector( busWidth - 1 downto 0 ) );

  end component mux2;

  component mux4 is

    generic ( busWidth : integer := 4 );

    port ( ip0 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip1 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip2 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip3 : in  std_logic_vector( busWidth - 1 downto 0 );
           sel : in  std_logic_vector( 1 downto 0 );
           op  : out std_logic_vector( busWidth - 1 downto 0 ) );

  end component mux4;

  component mux8 is

    generic ( busWidth : integer := 1 );

    port ( ip0 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip1 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip2 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip3 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip4 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip5 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip6 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip7 : in  std_logic_vector( busWidth - 1 downto 0 );
           sel : in  std_logic_vector( 2 downto 0 );
           op  : out std_logic_vector( busWidth - 1 downto 0 ) );

  end component mux8;

  component mux16 is

    generic ( busWidth : integer := 16 );

    port ( ip0  : in  std_logic_vector( busWidth - 1 downto 0 );
           ip1  : in  std_logic_vector( busWidth - 1 downto 0 );
           ip2  : in  std_logic_vector( busWidth - 1 downto 0 );
           ip3  : in  std_logic_vector( busWidth - 1 downto 0 );
           ip4  : in  std_logic_vector( busWidth - 1 downto 0 );
           ip5  : in  std_logic_vector( busWidth - 1 downto 0 );
           ip6  : in  std_logic_vector( busWidth - 1 downto 0 );
           ip7  : in  std_logic_vector( busWidth - 1 downto 0 );
           ip8  : in  std_logic_vector( busWidth - 1 downto 0 );
           ip9  : in  std_logic_vector( busWidth - 1 downto 0 );
           ip10 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip11 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip12 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip13 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip14 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip15 : in  std_logic_vector( busWidth - 1 downto 0 );
           sel  : in  std_logic_vector( 3 downto 0 );
           op   : out std_logic_vector( busWidth - 1 downto 0 ) );

  end component mux16;

  component decoder16 is

    port ( input  : in  std_logic_vector( 3 downto 0 );
           output : out std_logic_vector( 15 downto 0 ) );

  end component decoder16;

  component reg is

    generic ( busWidth : integer := 4 );

    port ( d      : in  std_logic_vector( busWidth - 1 downto 0 );
           clock  : in  std_logic;
```

```
        nReset : in  std_logic;
        enable : in  std_logic;
        q      : out std_logic_vector( busWidth - 1 downto 0 ) );

  end component reg;

  component tri is

    generic ( busWidth : integer := 16 );

    port ( input  : in  std_logic_vector( busWidth - 1 downto 0 );
           enable : in  std_logic;
           output : out std_logic_vector( busWidth - 1 downto 0 ) );

  end component tri;

  component bidirectional is

    generic ( busWidth : integer := 16 );

    port ( internal  : inout std_logic_vector( busWidth - 1 downto 0 );
           direction : in    std_logic;
           external  : inout std_logic_vector( busWidth - 1 downto 0 ) );

  end component bidirectional;

  component dff is

    generic ( resetValue : integer := 0 );

    port ( d      : in  std_logic;
           clock  : in  std_logic;
           nReset : in  std_logic;
           enable : in  std_logic;
           q      : out std_logic );

  end component dff;


end logicPackage;
```

## Listing C.15: Bidirectional Driver

```
-- $Id: bidirectional.vhd,v 1.1 2002/08/16 16:30:03 dillen Exp dillen $
--
-- $Log: bidirectional.vhd,v $
-- Revision 1.1  2002/08/16 16:30:03  dillen
-- Initial revision
--
--

library IEEE;
use IEEE.std_logic_1164.all;

library LOGIC;

entity bidirectional is

  -- synopsys template
  generic ( busWidth : integer := 16 );

  port ( internal  : inout std_logic_vector( busWidth - 1 downto 0 );
         direction : in    std_logic;
         external  : inout std_logic_vector( busWidth - 1 downto 0 ) );

end entity bidirectional;

architecture RTL of bidirectional is

  signal nDirection : std_logic;

  component tri is

    generic ( busWidth : integer := 16 );

    port ( input  : in  std_logic_vector( busWidth - 1 downto 0 );
           enable : in  std_logic;
           output : out std_logic_vector( busWidth - 1 downto 0 ) );

  end component tri;

begin

  nDirection <= not direction;

  U0 : tri generic map ( busWidth => busWidth )
           port    map ( input    => internal,
                         enable   => direction,
                         output   => external );
```

```
  U1 : tri generic map ( busWidth => busWidth )
           port    map ( input    => external,
                         enable   => nDirection,
                         output   => internal );

end architecture RTL;
```

## Listing C.16: 16-Bit Decoder

```
--
-- decoder16.vhd
--
-- Steve Dillen
-- Jun 18, 2002
--
-- $Id: decoder16.vhd,v 1.1 2002/07/17 17:27:34 dillen Exp $
--
-- $Log: decoder16.vhd,v $
-- Revision 1.1  2002/07/17 17:27:34  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

entity decoder16 is

  port ( input  : in  std_logic_vector( 3 downto 0 );
         output : out std_logic_vector( 15 downto 0 ) );

end decoder16;

architecture RTL of decoder16 is
begin

  op : process( input )
  begin

    case input is

      when "0000" => output <= X"0001";
      when "0001" => output <= X"0002";
      when "0010" => output <= X"0004";
      when "0011" => output <= X"0008";
      when "0100" => output <= X"0010";
      when "0101" => output <= X"0020";
      when "0110" => output <= X"0040";
      when "0111" => output <= X"0080";
      when "1000" => output <= X"0100";
      when "1001" => output <= X"0200";
      when "1010" => output <= X"0400";
      when "1011" => output <= X"0800";
      when "1100" => output <= X"1000";
      when "1101" => output <= X"2000";
      when "1110" => output <= X"4000";
      when "1111" => output <= X"8000";
      when others => output <= X"0000";

    end case;

  end process op;

end RTL;
```

## Listing C.17: D Flip-Flop

```
--
-- dff.vhd
--
-- Steve Dillen
-- March 13, 2003
--
-- $Id: dff.vhd,v 1.1 2003/03/14 18:04:15 dillen Exp dillen $
--
-- $Log: dff.vhd,v $
-- Revision 1.1  2003/03/14 18:04:15  dillen
-- Initial revision
--
--

library IEEE;
use IEEE.std_logic_1164.all;

entity dff is

  -- synopsys template
  generic ( resetValue : integer := 0 );
```

```
  port ( d      : in  std_logic;
         clock  : in  std_logic;
         nReset : in  std_logic;
         enable : in  std_logic;
         q      : out std_logic );

end dff;

architecture RTL of dff is
begin

  flipflop : process( nReset, clock )
  begin

    if (nReset = '0') then

      if (resetValue = 0) then

        q <= '0';

      else

        q <= '1';

      end if;

    elsif (clock = '1' and clock'event) then

      if (enable = '1') then

        q <= d;

      end if;

    end if;

  end process flipflop;

end RTL;
```

Listing C.18: 16-1 MUX

```
--
-- mux16.vhd
--
-- Steve Dillen
-- Jun 18, 2002
--
-- $Id: mux16.vhd,v 1.1 2002/07/17 17:27:34 dillen Exp dillen $
--
-- $Log: mux16.vhd,v $
-- Revision 1.1  2002/07/17 17:27:34  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

entity mux16 is

  -- synopsys template
  generic ( busWidth : integer := 16 );

  port ( ip0  : in  std_logic_vector( busWidth - 1 downto 0 );
         ip1  : in  std_logic_vector( busWidth - 1 downto 0 );
         ip2  : in  std_logic_vector( busWidth - 1 downto 0 );
         ip3  : in  std_logic_vector( busWidth - 1 downto 0 );
         ip4  : in  std_logic_vector( busWidth - 1 downto 0 );
         ip5  : in  std_logic_vector( busWidth - 1 downto 0 );
         ip6  : in  std_logic_vector( busWidth - 1 downto 0 );
         ip7  : in  std_logic_vector( busWidth - 1 downto 0 );
         ip8  : in  std_logic_vector( busWidth - 1 downto 0 );
         ip9  : in  std_logic_vector( busWidth - 1 downto 0 );
         ip10 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip11 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip12 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip13 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip14 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip15 : in  std_logic_vector( busWidth - 1 downto 0 );
         sel  : in  std_logic_vector( 3 downto 0 );
         op   : out std_logic_vector( busWidth - 1 downto 0 ) );

end mux16;

architecture RTL of mux16 is
begin

  mux : process( ip0, ip1, ip2, ip3, ip4, ip5, ip6, ip7,
                 ip8, ip9, ip10, ip11, ip12, ip13, ip14, ip15, sel )
  begin
```

```
    case sel is

      when "0000"  => op <= ip0;
      when "0001"  => op <= ip1;
      when "0010"  => op <= ip2;
      when "0011"  => op <= ip3;
      when "0100"  => op <= ip4;
      when "0101"  => op <= ip5;
      when "0110"  => op <= ip6;
      when "0111"  => op <= ip7;
      when "1000"  => op <= ip8;
      when "1001"  => op <= ip9;
      when "1010"  => op <= ip10;
      when "1011"  => op <= ip11;
      when "1100"  => op <= ip12;
      when "1101"  => op <= ip13;
      when "1110"  => op <= ip14;
      when "1111"  => op <= ip15;
      when others => op <= ip0;

    end case;

  end process mux;

end RTL;
```

Listing C.19: 8-1 MUX

```
--
-- mux8.vhd
--
-- Steve Dillen
-- Jun 7, 2002
--
-- $Id: mux8.vhd,v 1.1 2002/07/17 17:27:34 dillen Exp dillen $
--
-- $Log: mux8.vhd,v $
-- Revision 1.1  2002/07/17 17:27:34  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

library LOGIC;

entity mux8 is

  -- synopsys template
  generic ( busWidth : integer := 1 );

  port ( ip0 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip1 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip2 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip3 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip4 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip5 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip6 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip7 : in  std_logic_vector( busWidth - 1 downto 0 );
         sel : in  std_logic_vector( 2 downto 0 );
         op  : out std_logic_vector( busWidth - 1 downto 0 ) );

end mux8;

architecture RTL of mux8 is

  component mux2 is

    generic ( busWidth : integer := 1 );

    port ( ip0 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip1 : in  std_logic_vector( busWidth - 1 downto 0 );
           sel : in  std_logic;
           op  : out std_logic_vector( busWidth - 1 downto 0 ) );

  end component mux2;

  component mux4 is

    generic ( busWidth : integer := 1 );

    port ( ip0 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip1 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip2 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip3 : in  std_logic_vector( busWidth - 1 downto 0 );
           sel : in  std_logic_vector( 1 downto 0 );
           op  : out std_logic_vector( busWidth - 1 downto 0 ) );

  end component mux4;
```

```
  signal wire1 : std_logic_vector( busWidth - 1 downto 0 );
  signal wire2 : std_logic_vector( busWidth - 1 downto 0 );

begin

  U1 : mux4 generic map ( busWidth => 1 )
            port    map ( ip0      => ip0,
                          ip1      => ip1,
                          ip2      => ip2,
                          ip3      => ip3,
                          sel      => sel( 1 downto 0 ),
                          op       => wire1 );

  U2 : mux4 generic map ( busWidth => 1 )
            port    map ( ip0      => ip4,
                          ip1      => ip5,
                          ip2      => ip6,
                          ip3      => ip7,
                          sel      => sel( 1 downto 0 ),
                          op       => wire2 );

  U3 : mux2 generic map ( busWidth => 1 )
            port    map ( ip0      => wire1,
                          ip1      => wire2,
                          sel      => sel(2),
                          op       => op );

end RTL;
```

## Listing C.20: 4-1 MUX

```
--
-- mux4.vhd
--
-- Steve Dillen
-- Jun 7, 2002
--
-- $Id: mux4.vhd,v 1.1 2002/07/17 17:27:34 dillen Exp dillen $
--
-- $Log: mux4.vhd,v $
-- Revision 1.1  2002/07/17 17:27:34  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

library LOGIC;

entity mux4 is

  -- synopsys template
  generic ( busWidth : integer := 4 );

  port ( ip0 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip1 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip2 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip3 : in  std_logic_vector( busWidth - 1 downto 0 );
         sel : in  std_logic_vector( 1 downto 0 );
         op  : out std_logic_vector( busWidth - 1 downto 0 ) );

end mux4;

architecture RTL of mux4 is

  component mux2 is

    generic ( busWidth : integer := 4 );

    port ( ip0 : in  std_logic_vector( busWidth - 1 downto 0 );
           ip1 : in  std_logic_vector( busWidth - 1 downto 0 );
           sel : in  std_logic;
           op  : out std_logic_vector( busWidth - 1 downto 0 ) );

  end component mux2;

  signal wire1 : std_logic_vector( busWidth - 1 downto 0 );
  signal wire2 : std_logic_vector( busWidth - 1 downto 0 );

begin

  U0 : mux2 generic map ( busWidth => busWidth )
            port    map ( ip0      => ip0,
                          ip1      => ip1,
                          sel      => sel(0),
                          op       => wire1 );

  U1 : mux2 generic map ( busWidth => busWidth )
            port    map ( ip0      => ip2,
```

```
                        ip1       => ip3,
                        sel       => sel(0),
                        op        => wire2 );

  U2 : mux2 generic map ( busWidth => busWidth )
            port    map ( ip0      => wire1,
                        ip1       => wire2,
                        sel       => sel(1),
                        op        => op );

end RTL;
```

## Listing C.21: 2-1 MUX

```
--
-- mux2.vhd
--
-- Steve Dillen
-- May 21, 2002
--
-- $Id: mux2.vhd,v 1.1 2002/07/17 17:27:34 dillen Exp dillen $
--
-- $Log: mux2.vhd,v $
-- Revision 1.1  2002/07/17 17:27:34  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

entity mux2 is

  -- synopsys template
  generic ( busWidth : integer := 4 );

  port ( ip0 : in  std_logic_vector( busWidth - 1 downto 0 );
         ip1 : in  std_logic_vector( busWidth - 1 downto 0 );
         sel : in  std_logic;
         op  : out std_logic_vector( busWidth - 1 downto 0 ) );

end mux2;

architecture RTL of mux2 is
begin

  mux : process( ip0, ip1, sel )
  begin

    case sel is

      when '0'    => op <= ip0;
      when '1'    => op <= ip1;
      when others => op <= ip0;

    end case;

  end process mux;

end RTL;
```

## Listing C.22: Register

```
--
-- reg.vhd
--
-- Steve Dillen
-- Jun 7, 2002
--
-- $Id: reg.vhd,v 1.1 2002/07/17 17:27:34 dillen Exp dillen $
--
-- $Log: reg.vhd,v $
-- Revision 1.1  2002/07/17 17:27:34  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

entity reg is

  -- synopsys template
  generic ( busWidth : integer := 16 );
  port ( d      : in  std_logic_vector( busWidth - 1 downto 0 );
         clock  : in  std_logic;
         nReset : in  std_logic;
         enable : in  std_logic;
         q      : out std_logic_vector( busWidth - 1 downto 0 ) );
```

```
end reg;

architecture RTL of reg is

begin

  flipflop : process( nReset, clock )
  begin

    if ( nReset = '0' ) then

      q <= (others => '0');

    elsif ( clock = '1' and clock'event ) then

      if ( enable = '1' ) then

        q <= d;

      end if;

    end if;

  end process flipflop;

end RTL;
```

Listing C.23: Tri-state Buffer

```
-- $Id: tri.vhd,v 1.1 2002/07/17 17:27:34 dillen Exp dillen $
--
-- $Log: tri.vhd,v $
-- Revision 1.1  2002/07/17 17:27:34  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

entity tri is

  -- synopsys template
  generic ( busWidth : integer := 16 );

  port ( input  : in  std_logic_vector( busWidth - 1 downto 0 );
         enable : in  std_logic;
         output : out std_logic_vector( busWidth - 1 downto 0 ) );

end entity tri;

architecture RTL of tri is
begin

  buf : process( input, enable ) is
  begin

    if enable = '1' then

      output <= input;

    else

      output <= ( others => 'Z' );

    end if;

  end process buf;

end architecture RTL;
```

## C.2.2 DSP-RAM Processing Element Components

Listing C.24: ALU

```
--
-- alu.vhd
--
-- Steve Dillen
-- Jun 7, 2002
--
-- $Id: alu.vhd,v 1.3 2002/08/09 21:12:44 dillen Exp dillen $
--
-- $Log: alu.vhd,v $
-- Revision 1.3  2002/08/09 21:12:44  dillen
-- Final release for version 1
--
-- Revision 1.2  2002/07/17 19:50:33  dillen
```

```
-- Made generic widths to ALU
--
-- Revision 1.1  2002/07/17 17:16:47  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

library DWARE;
use DWARE.DWpackages.all;

library DW01;
use DW01.DW01_components.all;

library DW02;
use DW02.DW02_components.all;

library LOGIC;
use LOGIC.logicPackage.all;

entity alu is

  generic ( dw  : integer := 16;
            nsv : integer := 8; -- (4 or 8 only)
            sv0 : integer := 1;
            sv1 : integer := 2;
            sv2 : integer := 4;
            sv3 : integer := 6;
            sv4 : integer := 8;
            sv5 : integer := 16;
            sv6 : integer := 24;
            sv7 : integer := 32;
            ow  : integer := 48 );

  port ( inputA             : in  std_logic_vector( dw - 1 downto 0 );
         inputB             : in  std_logic_vector( dw - 1 downto 0 );
         accumulatorInput   : in  std_logic_vector( ow - 1 downto 0 );
         adderEnable        : in  std_logic;
         signControl        : in  std_logic;
         shiftValue         : in  std_logic_vector( 2 downto 0 );
         shiftMode          : in  std_logic_vector( 1 downto 0 );
         clock              : in  std_logic;
         enable             : in  std_logic;
         nReset             : in  std_logic;
         shiftedProductSum  : out std_logic_vector( ow - 1 downto 0 );
         inputAOut          : out std_logic_vector( dw - 1 downto 0 ) );

end alu;

architecture structural of alu is

  component shifter is

    generic ( dw  : integer := dw;
              nsv : integer := nsv; -- (4 or 8 only)
              sv0 : integer := sv0;
              sv1 : integer := sv1;
              sv2 : integer := sv2;
              sv3 : integer := sv3;
              sv4 : integer := sv4;
              sv5 : integer := sv5;
              sv6 : integer := sv6;
              sv7 : integer := sv7 );

    port ( dataIn     : in  std_logic_vector( dw - 1 downto 0 );
           shiftValue : in  std_logic_vector( 2 downto 0 );
           shiftMode  : in  std_logic_vector( 1 downto 0 );
           dataOut    : out std_logic_vector( dw - 1 downto 0 ) );

  end component shifter;

  signal inputAReg         : std_logic_vector( dw - 1 downto 0 );
  signal inputBReg         : std_logic_vector( dw - 1 downto 0 );
  signal productReg        : std_logic_vector( (dw * 2) - 1 downto 0 );
  signal internalAccInput  : std_logic_vector( ow - 1 downto 0 );
  signal internalEnable    : std_logic_vector( ow - 1 downto 0 );
  signal adderInput        : std_logic_vector( ow - 1 downto 0 );
  signal productSum        : std_logic_vector( ow - 1 downto 0 );
  signal logic0            : std_logic;
  signal notConnected      : std_logic;

  signal zeros             : std_logic_vector( ow - 1 downto 0 );
  signal signBits          : std_logic_vector( ow - 1 downto 0 );

begin

  -- Define some useful signals
  logic0 <= '0';

  -- Send the A register output out of the ALU
  inputAOut <= inputAReg;
```

```
-- Input Registers
aReg : reg generic map ( busWidth => dw )
           port    map ( d        => inputA,
                         clock    => clock,
                         nReset   => nReset,
                         enable   => enable,
                         q        => inputAReg );

bReg : reg generic map ( busWidth => dw )
           port    map ( d        => inputB,
                         clock    => clock,
                         nReset   => nReset,
                         enable   => enable,
                         q        => inputBReg );

-- Pipelined Multiplier
mult : DW02_mult_2_stage generic map ( A_width => dw,
                                       B_width => dw )
                         port    map ( A        => inputAReg,
                                       B        => inputBReg,
                                       TC       => signControl,
                                       CLK      => clock,
                                       PRODUCT  => productReg );

-- Product register
pReg : reg generic map ( busWidth => dw * 2 )
           port    map ( d        => productReg,
                         clock    => clock,
                         nReset   => nReset,
                         enable   => enable,
                         q        => adderInput( (dw * 2) - 1 downto 0 ) );

signBits <= (others => adderInput( dw * 2 - 1 ));

-- Sign extend adderInput
adderInput( ow - 1 downto dw * 2 ) <=
                                    signBits( ow - 1 downto dw * 2 )
                                    when signControl = '1'
                                    else zeros( ow - 1 downto dw * 2 );

-- Set the Accumulator input to 0 if adder is not enabled
internalEnable <= ( others => adderEnable );
internalAccInput <= accumulatorInput and internalEnable;

-- Adder
adder : DW01_add generic map ( width => ow )
                port    map ( A     => internalAccInput,
                              B     => adderInput,
                              CI    => logic0,
                              SUM   => productSum,
                              CO    => notConnected );

shift : shifter generic map ( dw  => ow,
                              nsv => nsv,
                              sv0 => sv0,
                              sv1 => sv1,
                              sv2 => sv2,
                              sv3 => sv3,
                              sv4 => sv4,
                              sv5 => sv5,
                              sv6 => sv6,
                              sv7 => sv7 )

              port    map ( dataIn     => productSum,
                            shiftValue => shiftValue,
                            shiftMode  => shiftMode,
                            dataOut    => shiftedProductSum );

end structural;
```

## Listing C.25: Arbiter

```
--
-- $Id: arbiter.vhd,v 1.3 2003/03/14 18:42:49 dillen Exp $
--
-- $Log: arbiter.vhd,v $
-- Revision 1.3  2003/03/14 18:42:49  dillen
-- Working copy of the arbiter
--
-- Revision 1.2  2003/03/10 18:13:35  dillen
-- Tested arbiter
--
-- Revision 1.1  2002/08/09 21:14:14  dillen
-- Initial revision
--
--

library IEEE;
use IEEE.std_logic_1164.all;
```

```
library DWARE;
use DWARE.DWpackages.all;

library DW01;
use DW01.DW01_components.all;

library DW03;
use DW03.DW03_components.all;

entity arbiter is

  generic ( busDepth : integer := 4 );

  port ( broadcastBusRegister : in  std_logic_vector( 2 ** busDepth - 1 downto 0 );
         broadcastBus         : in  std_logic_vector( 2 ** busDepth - 1 downto 0 );
         clock                : in  std_logic;
         enable               : in  std_logic;
         nReset               : in  std_logic;
         win                  : out std_logic );

end entity arbiter;

architecture mixed of arbiter is

  -- State machine types
  type states is ( initial, enabled, disabled );
  signal currentState : states;
  signal nextState    : states;

  -- Counter values
  signal resetCount : std_logic_vector( busDepth - 1 downto 0 );
  signal bitCount   : std_logic_vector( busDepth - 1 downto 0 );

  -- Control signals
  signal logicOne      : std_logic;
  signal downCounter   : std_logic;
  signal notConnected  : std_logic;
  signal nLoadCounter  : std_logic;

  -- Bits to compare
  signal bbRegisterBit : std_logic_vector( 0 downto 0 );
  signal bbBit         : std_logic_vector( 0 downto 0 );

  -- Match detected
  signal match         : std_logic;

begin

  -- Wire the initial signals and control signals
  logicOne     <= '1';
  downCounter  <= '0';
  resetCount   <= (others => '1');

  -- Configure a down counter for the bit number
  bitCounter : DW03_updn_ctr generic map ( width => busDepth )
                             port    map ( data  => resetCount,
                                           up_dn => downCounter,
                                           load  => nLoadCounter,
                                           cen   => logicOne,
                                           clk   => clock,
                                           reset => logicOne,
                                           count => bitCount,
                                           tercnt => notConnected );

  -- Select the bits from BBREG
  bbregSelect : DW01_mux_any generic map ( A_width   => 2 ** busDepth,
                                           SEL_width => busDepth,
                                           MUX_width => 1 )
                             port    map ( A         => broadcastBusRegister,
                                           SEL       => bitCount,
                                           MUX       => bbRegisterBit );

  -- Select the bits from the broadcast bus
  bbSelect : DW01_mux_any generic map ( A_width   => 2 ** busDepth,
                                        SEL_width => busDepth,
                                        MUX_width => 1 )
                          port    map ( A         => broadcastBus,
                                        SEL       => bitCount,
                                        MUX       => bbBit );

  -- Compare the bits and select the win signal
  match <= bbRegisterBit(0) xnor bbBit(0);

  -- Next state logic
  nextStateLogic : process ( match, enable, currentState ) is
  begin

    case currentState is

      when initial =>
```

```
        -- Load counter, and don't drive the broadcast bus.
        nLoadCounter <= '0';

        -- Stay in the initial state until enabled
        if enable = '0' then

          nextState <= initial;

        else

          nextState <= enabled;

        end if;

      when enabled =>

        -- While enabled, drive the broadcast bus, and don't load
        -- the default count value
        nLoadCounter <= '1';

        -- If enable goes low, go back to initial state
        if enable = '0' then

          nextState <= initial;

        else

          -- If there is a match, stay enabled, otherwise go to
          -- the disabled state
          if match = '1' then

            nextState <= enabled;

          else

            nextState <= disabled;

          end if;

        end if;

      when disabled =>

        -- While disabled, don't drive the broadcast bus and load
        -- the counter with the initial value
        nLoadCounter <= '0';

        -- If enable goes low, go back to the initial state, otherwise
        -- stay disabled
        if enable = '0' then

          nextState <= initial;

        else

          nextState <= disabled;

        end if;

    end case;

  end process nextStateLogic;

  registerState : process ( clock, nReset ) is
  begin

    if nReset = '0' then

      currentState <= initial;

    elsif rising_edge( clock ) then

      currentState <= nextState;

    end if;

  end process registerState;

  -- WIN == nLoadCounter
  win <= nLoadCounter;

end architecture mixed;
```

Listing C.26: Byte Select

```
--
-- byteSelector.vhd
--
-- Steve Dillen
-- Jun 12, 2002
```

```
--
-- $Id: byteSelector.vhd,v 1.4 2003/02/14 04:54:13 dillen Exp $
--
-- $Log: byteSelector.vhd,v $
-- Revision 1.4  2003/02/14 04:54:13  dillen
-- Merged into PE directory
--
-- Revision 1.3  2002/08/09 21:15:27  dillen
-- Final release for version 1
--
-- Revision 1.2  2002/07/17 20:08:28  dillen
-- Made a generic bus width for the byte selector
--
-- Revision 1.1  2002/07/17 17:22:13  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

library LOGIC;
use LOGIC.logicPackage.all;

entity byteSelector is

  generic ( busWidth : integer := 16 );

  port ( dataIn         : in  std_logic_vector( busWidth - 1 downto 0 );
         byteSelection  : in  std_logic;
         enable         : in  std_logic;
         dataOut        : out std_logic_vector( busWidth - 1 downto 0 ) );

end byteSelector;

architecture RTL of byteSelector is

  signal internalInput : std_logic_vector( busWidth - 1 downto 0 );
  signal zeros         : std_logic_vector( busWidth - 1 downto 0 );

begin

  zeros <= (others => '0');

  -- Zeros for the MSB
  internalInput( busWidth - 1 downto busWidth / 2 ) <= zeros( busWidth - 1 downto busWidth / 2 );

  -- Select the byte
  level1 : mux2 generic map ( busWidth => busWidth / 2 )
                port    map ( ip0      => dataIn( (busWidth / 2) - 1 downto 0 ),
                              ip1      => dataIn( busWidth - 1 downto busWidth/2 ),
                              sel      => byteSelection,
                              op       => internalInput( (busWidth / 2) - 1 downto 0 )
                            );

  -- Enable the byte select hardware
  level2 : mux2 generic map ( busWidth => busWidth )
                port    map ( ip0      => dataIn,
                              ip1      => internalInput,
                              sel      => enable,
                              op       => dataOut
                            );

end RTL;
```

## Listing C.27: Shifter

```
--
-- shifter.vhd
--
-- Steve Dillen
-- Jun 10, 2002
--
-- $Id: shifter.vhd,v 1.7 2003/02/14 04:54:50 dillen Exp $
--
-- $Log: shifter.vhd,v $
-- Revision 1.7  2003/02/14 04:54:50  dillen
-- Removed shiftTypes and merged into PE directory
--
-- Revision 1.6  2002/08/09 21:19:21  dillen
-- Final release for version 1
--
-- Revision 1.5  2002/07/17 23:53:19  dillen
-- Modified to remove number of shift values (fixed at 8)
--
-- Revision 1.4  2002/07/17 22:10:10  dillen
-- More generic version (generic value array)
--
-- Revision 1.3  2002/07/17 21:33:17  dillen
-- Fixed the model for synthesis problems (others => '1')
--
```

```
-- Revision 1.2  2002/07/17 20:53:21  dillen
-- Made generic shift values and bus width
--
-- Revision 1.1  2002/07/17 17:34:30  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

entity shifter is

  generic ( dw  : integer := 48;   -- Data bus width
            nsv : integer := 4;    -- Number of shift values (4 or 8)
            sv0 : integer := 1;    -- Shift value 0
            sv1 : integer := 2;    -- Shift value 1
            sv2 : integer := 4;    -- Shift value 2
            sv3 : integer := 6;    -- Shift value 3
            sv4 : integer := 8;    -- Shift value 4 (sv4-7 not used
            sv5 : integer := 16;   -- Shift value 5  is nsv = 4)
            sv6 : integer := 24;   -- Shift value 6
            sv7 : integer := 32 ); -- Shift value 7

  port ( dataIn     : in  std_logic_vector( dw - 1 downto 0 );
         shiftValue : in  std_logic_vector( 2 downto 0 );
         shiftMode  : in  std_logic_vector( 1 downto 0 );
         dataOut    : out std_logic_vector( dw - 1 downto 0 ) );

end entity shifter;

architecture RTL of shifter is

  -- Constants used in code
  constant NoShift : std_logic_vector( 1 downto 0 ) := "00";
  constant LSL     : std_logic_vector( 1 downto 0 ) := "01";
  constant LSR     : std_logic_vector( 1 downto 0 ) := "10";
  constant ASR     : std_logic_vector( 1 downto 0 ) := "11";

  -- Signals
  signal data : std_logic_vector( dw - 1 downto 0 );

begin

  data <= (others => dataIn( dw - 1 )) when shiftMode = ASR else
          (others => '0');

  shifter4 : if nsv = 4 generate

    decodeShiftMode4 : process( shiftValue, dataIn, shiftMode, data )
    begin

      case shiftMode is

        when NoShift =>

          dataOut <= dataIn;

        when LSL =>

          case shiftValue is

            when "000" =>

              dataOut( dw - 1  downto sv0 ) <= dataIn( dw - sv0 - 1 downto 0 );
              dataOut( sv0 - 1 downto 0 )   <= data( sv0 - 1 downto 0 );

            when "001" =>

              dataOut( dw - 1  downto sv1 ) <= dataIn( dw - sv1 - 1 downto 0 );
              dataOut( sv1 - 1 downto 0 )   <= data( sv1 - 1 downto 0 );

            when "010" =>

              dataOut( dw - 1  downto sv2 ) <= dataIn( dw - sv2 - 1 downto 0 );
              dataOut( sv2 - 1 downto 0 )   <= data( sv2 - 1 downto 0 );

            when "011" =>

              dataOut( dw - 1  downto sv3 ) <= dataIn( dw - sv3 - 1 downto 0 );
              dataOut( sv3 - 1 downto 0 )   <= data( sv3 - 1 downto 0 );

            when others =>

              dataOut <= dataIn;

          end case;

        when LSR | ASR =>

          case shiftValue is

            when "000" =>
```

```
            dataOut( dw - sv0 - 1 downto 0 )  <= dataIn( dw - 1 downto sv0 );
            dataOut( dw - 1 downto dw - sv0 ) <= data( dw - 1 downto dw - sv0 );

          when "001" =>

            dataOut( dw - sv1 - 1 downto 0 )  <= dataIn( dw - 1 downto sv1 );
            dataOut( dw - 1 downto dw - sv1 ) <= data( dw - 1 downto dw - sv1 );

          when "010" =>

            dataOut( dw - sv2 - 1 downto 0 )  <= dataIn( dw - 1 downto sv2 );
            dataOut( dw - 1 downto dw - sv2 ) <= data( dw - 1 downto dw - sv2 );

          when "011" =>

            dataOut( dw - sv3 - 1 downto 0 )  <= dataIn( dw - 1 downto sv3 );
            dataOut( dw - 1 downto dw - sv3 ) <= data( dw - 1 downto dw - sv3 );

          when others =>

            dataOut <= dataIn;

        end case;

      when others =>

        dataOut <= dataIn;

    end case;

  end process decodeShiftMode4;

end generate shifter4;

shifter8 : if nsv = 8 generate

  decodeShiftMode : process( shiftValue, dataIn, shiftMode, data )
  begin

    case shiftMode is

      when NoShift =>

        dataOut <= dataIn;

      when LSL =>

        case shiftValue is

          when "000" =>

            dataOut( dw - 1 downto sv0 ) <= dataIn( dw - sv0 - 1 downto 0 );
            dataOut( sv0 - 1 downto 0 )  <= data( sv0 - 1 downto 0 );

          when "001" =>

            dataOut( dw - 1 downto sv1 ) <= dataIn( dw - sv1 - 1 downto 0 );
            dataOut( sv1 - 1 downto 0 )  <= data( sv1 - 1 downto 0 );

          when "010" =>

            dataOut( dw - 1 downto sv2 ) <= dataIn( dw - sv2 - 1 downto 0 );
            dataOut( sv2 - 1 downto 0 )  <= data( sv2 - 1 downto 0 );

          when "011" =>

            dataOut( dw - 1 downto sv3 ) <= dataIn( dw - sv3 - 1 downto 0 );
            dataOut( sv3 - 1 downto 0 )  <= data( sv3 - 1 downto 0 );

          when "100" =>

            dataOut( dw - 1 downto sv4 ) <= dataIn( dw - sv4 - 1 downto 0 );
            dataOut( sv4 - 1 downto 0 )  <= data( sv4 - 1 downto 0 );

          when "101" =>

            dataOut( dw - 1 downto sv5 ) <= dataIn( dw - sv5 - 1 downto 0 );
            dataOut( sv5 - 1 downto 0 )  <= data( sv5 - 1 downto 0 );

          when "110" =>

            dataOut( dw - 1 downto sv6 ) <= dataIn( dw - sv6 - 1 downto 0 );
            dataOut( sv6 - 1 downto 0 )  <= data( sv6 - 1 downto 0 );

          when "111" =>

            dataOut( dw - 1 downto sv7 ) <= dataIn( dw - sv7 - 1 downto 0 );
            dataOut( sv7 - 1 downto 0 )  <= data( sv7 - 1 downto 0 );

          when others =>
```

```
            dataOut <= dataIn;

          end case;

        when LSR | ASR =>

          case shiftValue is

            when "000" =>

              dataOut( dw - sv0 - 1 downto 0 )  <= dataIn( dw - 1 downto sv0 );
              dataOut( dw - 1 downto dw - sv0 ) <= data( dw - 1 downto dw - sv0 );

            when "001" =>

              dataOut( dw - sv1 - 1 downto 0 )  <= dataIn( dw - 1 downto sv1 );
              dataOut( dw - 1 downto dw - sv1 ) <= data( dw - 1 downto dw - sv1 );

            when "010" =>

              dataOut( dw - sv2 - 1 downto 0 )  <= dataIn( dw - 1 downto sv2 );
              dataOut( dw - 1 downto dw - sv2 ) <= data( dw - 1 downto dw - sv2 );

            when "011" =>

              dataOut( dw - sv3 - 1 downto 0 )  <= dataIn( dw - 1 downto sv3 );
              dataOut( dw - 1 downto dw - sv3 ) <= data( dw - 1 downto dw - sv3 );

            when "100" =>

              dataOut( dw - sv4 - 1 downto 0 )  <= dataIn( dw - 1 downto sv4 );
              dataOut( dw - 1 downto dw - sv4 ) <= data( dw - 1 downto dw - sv4 );

            when "101" =>

              dataOut( dw - sv5 - 1 downto 0 )  <= dataIn( dw - 1 downto sv5 );
              dataOut( dw - 1 downto dw - sv5 ) <= data( dw - 1 downto dw - sv5 );

            when "110" =>

              dataOut( dw - sv6 - 1 downto 0 )  <= dataIn( dw - 1 downto sv6 );
              dataOut( dw - 1 downto dw - sv6 ) <= data( dw - 1 downto dw - sv6 );

            when "111" =>

              dataOut( dw - sv7 - 1 downto 0 )  <= dataIn( dw - 1 downto sv7 );
              dataOut( dw - 1 downto dw - sv7 ) <= data( dw - 1 downto dw - sv7 );

            when others =>

              dataOut <= dataIn;

          end case;

        when others =>

          dataOut <= dataIn;

      end case;

    end process decodeShiftMode;

  end generate shifter8;

end architecture RTL;
```

## Listing C.28: Stack Element

```
--
-- stackElement.vhd
--
-- Steve Dillen
--
-- Jun 11, 2002
--
-- $Id: stackElement.vhd,v 1.3 2003/03/14 18:43:16 dillen Exp $
--
-- $Log: stackElement.vhd,v $
-- Revision 1.3  2003/03/14 18:43:16  dillen
-- New version of the stack element
--
-- Revision 1.1  2002/07/17 17:36:43  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

library LOGIC;
use LOGIC.logicPackage.all;
```

```
entity stackElement is

  port ( data     : in  std_logic;
         previous : in  std_logic;
         opMode   : in  std_logic;
         enable   : in  std_logic;
         clock    : in  std_logic;
         nReset   : in  std_logic;
         q        : out std_logic );

end stackElement;

architecture RTL of stackElement is

  signal qNext     : std_logic;
  signal qInternal : std_logic;

begin

  -- Perform next state logic
  qNext <= ((opMode and (not qInternal)) and previous) or -- TOGGLE
           ((not opMode) and previous and data );         -- PUSH

  -- Map the flip flop as the main component
  flipflop : dff generic map ( resetValue => 0 )
                 port    map ( d      => qNext,
                               clock  => clock,
                               nReset => nReset,
                               enable => enable,
                               q      => qInternal );

  -- Connect the output
  q <= qInternal;

end RTL;
```

Listing C.29: Stack

```
--
-- stack.vhd
--
-- Steve Dillen
-- Jun 11, 2002
--
-- $Id: stack.vhd,v 1.4 2003/03/14 18:43:07 dillen Exp $
--
-- $Log: stack.vhd,v $
-- Revision 1.4  2003/03/14 18:43:07  dillen
-- New version of the stack
--
-- Revision 1.3  2002/08/09 21:20:21  dillen
-- Final release for version 1
--
-- Revision 1.2  2002/07/18 19:05:51  dillen
-- Made the stack depth generic
--
-- Revision 1.1  2002/07/17 17:36:43  dillen
-- Initial revision
--

library IEEE;
use IEEE.std_logic_1164.all;

library LOGIC;
use LOGIC.logicPackage.all;

-- Make it a stack of depth 4
entity stack is

  generic ( stackDepth : integer := 8 );

  port ( dataIn  : in  std_logic;
         clock   : in  std_logic;
         nReset  : in  std_logic;
         opMode  : in  std_logic_vector( 1 downto 0 );
         dataOut : out std_logic );

end stack;

architecture RTL of stack is

  component stackElement is

    port ( data     : in  std_logic;
           previous : in  std_logic;
           opMode   : in  std_logic;
           enable   : in  std_logic;
           clock    : in  std_logic;
           nReset   : in  std_logic;
```

```
        q         : out std_logic );

  end component stackElement;

  signal enables         : std_logic_vector( stackDepth - 1 downto 0 );
  signal stackOut        : std_logic_vector( stackDepth downto 0 );
  signal selectStackOut  : std_logic_vector( stackDepth downto 0 );

  signal stackAddress    : std_logic_vector( stackDepth + 2 downto 0 );
  signal shiftInput      : std_logic_vector( stackDepth + 1 downto 1 );

  -- Set the shift register control bits
  signal shiftRegisterEnable : std_logic;
  signal shiftDirection      : std_logic;

  -- opMode
  constant NOP    : std_logic_vector( 1 downto 0 ) := b"00";
  constant PUSH   : std_logic_vector( 1 downto 0 ) := b"01";
  constant POP    : std_logic_vector( 1 downto 0 ) := b"10";
  constant TOGGLE : std_logic_vector( 1 downto 0 ) := b"11";

begin

  -- StackOut(0) is always enabled
  stackOut(0) <= '1';

  -- Generate the stack
  stackElements : for i in 0 to stackDepth - 1 generate

    stackArray : stackElement port map ( data     => dataIn,
                                         previous => stackOut( i ),
                                         opMode   => opMode(1),
                                         enable   => enables( i ),
                                         clock    => clock,
                                         nReset   => nReset,
                                         q        => stackOut( i + 1)
                                       );

  end generate stackElements;

  -- Generate stack enables mux
  stackEnables : process( opMode, stackAddress ) is
  begin

    case opMode is

      when NOP    => enables <= (others => '0');
      when PUSH   => enables <= stackAddress( stackDepth downto 1 );
      when POP    => enables <= (others => '0');
      when TOGGLE => enables <= stackAddress( stackDepth + 1 downto 2 );
      when others => enables <= (others => '0');

    end case;

  end process stackEnables;

  -- Generate the output of the stack
  selectStackOut <= stackOut and stackAddress( stackDepth + 1 downto 1 );

  stackOutput : process( selectStackOut ) is

    variable i        : integer;
    variable globalOr : std_logic;

  begin

    globalOr := selectStackOut(1) or selectStackOut(0);

    if stackDepth > 1 then

      for i in 2 to stackDepth loop

        globalOr := globalOr or selectStackOut(i);

      end loop;

    end if;

    dataOut <= globalOr;

  end process stackOutput;


  -- Set first and last bits of the stack address
  stackAddress(0)               <= '0';
  stackAddress(stackDepth + 2) <= '0';

  -- When is the shift register enabled
  shiftRegisterEnable <= ((not opMode(1)) and opMode(0)) or  -- PUSH or
                         (opMode(1) and (not opMode(0)));    -- POP

  shiftDirection <= (not opMode(1)) and opMode(0); -- PUSH
```

```
  -- Create the shift register
  shiftRegister : for i in 1 to stackDepth + 1 generate

    muxedIn : mux2 generic map ( busWidth => 1 )
                   port    map ( ip0      => stackAddress( i + 1 downto i + 1),
                                 ip1      => stackAddress( i - 1 downto i - 1),
                                 sel      => shiftDirection,
                                 op       => shiftInput(i downto i) );

    firstFlop  : if i = 1 generate

      bit0 : dff generic map ( resetValue => 1 )
                 port    map ( d          => shiftInput(i),
                               clock      => clock,
                               nReset     => nReset,
                               enable     => shiftRegisterEnable,
                               q          => stackAddress(i) );

    end generate firstflop;

    otherFlops : if i > 1 generate

      bit1 : dff generic map ( resetValue => 0 )
                 port    map ( d          => shiftInput(i),
                               clock      => clock,
                               nReset     => nReset,
                               enable     => shiftRegisterEnable,
                               q          => stackAddress(i) );


    end generate otherFlops;

  end generate shiftRegister;

end RTL;
```

## Listing C.30: PE

```
--
-- pe.vhd
--
-- Steve Dillen
-- Jun. 12, 2002
--
-- $Id: pe.vhd,v 1.5 2003/02/14 04:54:27 dillen Exp dillen $
--
-- $Log: pe.vhd,v $
-- Revision 1.5  2003/02/14 04:54:27  dillen
-- Merged all but LOGIC into PE directory
--
-- Revision 1.4  2002/08/09 21:18:10  dillen
-- Final release for version 1
--
-- Revision 1.3  2002/07/09 17:48:03  dillen
-- Data buses now use inout types
--
-- Revision 1.2  2002/07/09 15:23:22  dillen
-- Modified Shift Bus to use inout's
--

library IEEE;
use IEEE.std_logic_1164.all;

library DWARE;
use DWARE.DWpackages.all;

library DW01;
use DW01.DW01_components.all;

library LOGIC;
use LOGIC.logicPackage.all;

entity pe is

  -- I normally don't use abbreviations since it makes
  -- very unreadable code, however the synthesis tool
  -- (FPGA Compiler II) creates file names that were
  -- WAY too long for Solaris and it couldn't even
  -- synthesize under Windows.  Also, FPGA Compiler II
  -- requires some tool to use custom types (such as
  -- an array for the shift values) which CMC didn't
  -- ship (that I can find) so I do not use any
  -- custom types.
  generic ( dw    : integer := 16;    -- Databus Width
            sbw   : integer := 48;    -- Shift Bus Width
                                      -- (Can be dw or 3*dw only)
            obits : integer := 16;    -- Number of overhead bits
            nsv   : integer := 8;     -- Number of shift values
```

```
                                        -- (4 or 8 only)
            sv0   : integer := 1;       -- Shift Value 0
            sv1   : integer := 2;       -- Shift Value 1
            sv2   : integer := 4;       -- Shift Value 2
            sv3   : integer := 6;       -- Shift Value 3
            sv4   : integer := 8;       -- Shift Value 4
            sv5   : integer := 16;      -- Shift Value 5
            sv6   : integer := 24;      -- Shift Value 6
            sv7   : integer := 32;      -- Shift Value 7
            isr   : integer := 1;       -- Include Shift Register Logic
            ia    : integer := 1;       -- Include Arbiter Logic
            sd    : integer := 8 );     -- Stack Depth

  port ( clock           : in    std_logic;
         nReset          : in    std_logic;
         controlWord     : in    std_logic_vector( 43 downto 0 );
         ramDataA        : inout std_logic_vector( dw - 1 downto 0 );
         ramDataB        : inout std_logic_vector( dw - 1 downto 0 );
         leftPE          : inout std_logic_vector( sbw - 1 downto 0 );
         rightPE         : inout std_logic_vector( sbw - 1 downto 0 );
         broadcastBus    : in    std_logic_vector( dw - 1 downto 0 );
         broadcastBusOut : out   std_logic_vector( dw - 1 downto 0 );
         ramEnable       : out   std_logic );

end entity pe;

architecture RTL of pe is

  component alu is

    generic ( dw  : integer := dw;
              nsv : integer := nsv; -- (4 or 8 only)
              sv0 : integer := sv0;
              sv1 : integer := sv1;
              sv2 : integer := sv2;
              sv3 : integer := sv3;
              sv4 : integer := sv4;
              sv5 : integer := sv5;
              sv6 : integer := sv6;
              sv7 : integer := sv7;
              ow  : integer := (dw * 2) + 8bits );

    port ( inputA             : in  std_logic_vector( dw - 1 downto 0 );
           inputB             : in  std_logic_vector( dw - 1 downto 0 );
           accumulatorInput   : in  std_logic_vector( ow - 1 downto 0 );
           adderEnable        : in  std_logic;
           signControl        : in  std_logic;
           shiftValue         : in  std_logic_vector( 2 downto 0 );
           shiftMode          : in  std_logic_vector( 1 downto 0 );
           clock              : in  std_logic;
           enable             : in  std_logic;
           nReset             : in  std_logic;
           shiftedProductSum  : out std_logic_vector( ow - 1 downto 0 );
           inputAOut          : out std_logic_vector( dw - 1 downto 0 ) );

  end component alu;

  component byteSelector is

    generic ( busWidth : integer := 16 );

    port ( dataIn        : in  std_logic_vector( busWidth - 1 downto 0 );
           byteSelection : in  std_logic;
           enable        : in  std_logic;
           dataOut       : out std_logic_vector( busWidth - 1 downto 0 ) );

  end component byteSelector;

  component stack is

    generic ( stackDepth : integer := sd );

    port ( dataIn  : in  std_logic;
           nReset  : in  std_logic;
           clock   : in  std_logic;
           opMode  : in  std_logic_vector( 1 downto 0 );
           dataOut : out std_logic );

  end component stack;

  component arbiter is

    generic ( busDepth : integer := 4 );
    port ( broadcastBusRegister : in  std_logic_vector( 2 ** busDepth - 1 downto 0 );
           broadcastBus         : in  std_logic_vector( 2 ** busDepth - 1 downto 0 );
           clock                : in  std_logic;
           enable               : in  std_logic;
           nReset               : in  std_logic;
           win                  : out std_logic );

  end component arbiter;
```

```
-- ALU Signals
signal aluInputA   : std_logic_vector( dw - 1 downto 0 );
signal aluInputB   : std_logic_vector( dw - 1 downto 0 );
signal aluEnable   : std_logic;
signal shifterOut  : std_logic_vector( (dw * 3) - 1 downto 0 );
signal inputAOut   : std_logic_vector( dw - 1 downto 0 );

-- ALU input signals
signal muxAOut : std_logic_vector( dw - 1 downto 0 );

-- Accumulator Signals
signal accumulatorIn       : std_logic_vector( (dw * 3) - 1 downto 0 );
signal accumulatorOut      : std_logic_vector( (dw * 3) - 1 downto 0 );
signal accumulator0Enable : std_logic;
signal accumulator1Enable : std_logic;
signal accumulator2Enable : std_logic;
signal nAccReset           : std_logic;

-- Shift register signals
signal shifterOutWords      : std_logic_vector( dw - 1 downto 0 );
signal shiftRegisterIn      : std_logic_vector( sbw - 1 downto 0 );
signal shiftRegisterOut     : std_logic_vector( sbw - 1 downto 0 );
signal shiftRegisterEnable : std_logic;

-- Broadcast Bus Signals
signal broadcastBusRegisterIn      : std_logic_vector( dw - 1 downto 0 );
signal broadcastBusRegisterOut     : std_logic_vector( dw - 1 downto 0 );
signal broadcastBusRegisterEnable : std_logic;

-- Bus Signals
signal leftBus     : std_logic_vector( sbw - 1 downto 0 );
signal rightBus    : std_logic_vector( sbw - 1 downto 0 );
signal interPE     : std_logic_vector( sbw - 1 downto 0 );
signal dataBus     : std_logic_vector( dw - 1 downto 0 );
signal dataBusA    : std_logic_vector( dw - 1 downto 0 );
signal dataBusB    : std_logic_vector( dw - 1 downto 0 );

-- Comparator Signals
constant LT : integer := 0;
constant GT : integer := 1;
constant EQ : integer := 2;
constant LE : integer := 3;
constant GE : integer := 4;
constant NE : integer := 5;
signal comparatorOut : std_logic_vector( 5 downto 0 );

-- Arbiter signals
signal win : std_logic_vector( 0 downto 0 );

-- Stack signals
signal stackInput  : std_logic_vector( 0 downto 0 );
signal stackOutput : std_logic;

-- 1 and 0
signal logicOne   : std_logic;
signal zeros      : std_logic_vector( sbw - 1 downto 0 );

begin

-- ControlWord breakdown:
-- 0       - Adder enable
-- 3-1     - Shift distance
-- 5-4     - Shift mode
-- 6       - ALU enable
-- 7       - Input A mux select
-- 8       - Byte select
-- 9       - Byte select enable
-- 10      - Input B mux select
-- IF sbw = dw
-- 11      - interPE mux select
-- 12      - Accumulator 0 mux select
-- 13      - Accumulator 1 mux select
-- 14      - Accumulator 2 mux select
-- 15      - Accumulator Register clear
-- 16      - Unused
-- ELSE
-- 12-11   - Accumulator 0 mux select
-- 14-13   - Accumulator 1 mux select
-- 16-15   - Accumulator 2 mux select
-- ENDIF
-- 17      - Accumulator 0 register enable
-- 18      - Accumulator 1 register enable
-- 19      - Accumulator 2 register enable
-- 21-20   - Broadcast bus mux select
-- 22      - Broadcast bus register enable
-- 25-23   - Stack mux select
-- 27-26   - Stack opmode
-- IF ia = 1 THEN
-- 28      - Arbiter enable
-- ELSE
-- 28      - Unused
-- ENDIF
```

```
-- 30-29  - Data bus mux
-- 31     - Bus A driver
-- 32     - Bus B driver
-- 33     - Accumulator left driver
-- 34     - Accumulator right driver
-- 35     - Left bus driver
-- 36     - Right bus driver
-- IF isr = 1 and sbw = dw * 3 THEN
-- 38-37  - Shift register mux select
-- 39     - Shift register enable
-- 40     - Shift register left driver
-- 41     - Shift register right driver
-- ELSIF isr = 1 and sbw = dw
-- 37     - Shift register mux select
-- 38     - Unused
-- 39     - Shift register enable
-- 40     - Shift register left driver
-- 41     - Shift register right driver
-- ELSE
-- 41-37  - Unused
-- ENDIF
-- IF sbw = dw * 3 THEN
-- 43-42  - Data bus B mux select
-- ELSE
-- 43-42  - Unused
-- ENDIF
logicOne   <= '1';
zeros      <= ( others => '0' );

-- Connect the ALU portion of the PE
aluEnable                  <= stackOutput and controlWord( 6 );

processor : alu generic map ( dw  => dw,
                              nsv => nsv,
                              sv0 => sv0,
                              sv1 => sv1,
                              sv2 => sv2,
                              sv3 => sv3,
                              sv4 => sv4,
                              sv5 => sv5,
                              sv6 => sv6,
                              sv7 => sv7,
                              ow  => dw * 2 + obits )

                    port    map ( inputA             => aluInputA,
                                  inputB             => aluInputB,
                                  accumulatorInput   => accumulatorOut( dw * 2 + obits - 1 downto 0 ),
                                  adderEnable        => controlWord(0),
                                  signControl        => logicOne,
                                  shiftValue         => controlWord( 3 downto 1 ),
                                  shiftMode          => controlWord( 5 downto 4 ),
                                  clock              => clock,
                                  enable             => aluEnable,
                                  nReset             => nReset,
                                  shiftedProductSum  => shifterOut( dw * 2 + obits - 1 downto 0 ),
                                  inputAOut          => inputAOut );

-- Sign extend the shifter word
signExtend : for i in dw * 2 + obits - 1 to dw * 3 - 1 generate

  shifterOut( i ) <= shifterOut( dw * 2 + obits - 1 );

end generate signExtend;

-- Connect the MUX's at the input to the ALU
inputMuxA : mux2 generic map ( busWidth => dw )
                 port    map ( ip0      => ramDataA,
                               ip1      => ramDataB,
                               sel      => controlWord(7),
                               op       => muxAOut );

-- Connect the byte Selector
byteSelect : byteSelector port map ( dataIn        => muxAOut,
                                     byteSelection => controlWord(8),
                                     enable        => controlWord(9),
                                     dataOut       => aluInputA
                                   );

inputMuxB : mux2 generic map ( busWidth => dw )
                 port    map ( ip0      => ramDataB,
                               ip1      => broadcastBusRegisterOut,
                               sel      => controlWord(10),
                               op       => aluInputB
                             );

shiftBus1 : if sbw = dw generate

  -- Create a MUX for the LR shift bus to accumulator register
  interPEMux : mux2 generic map ( busWidth => sbw )
                    port    map ( ip0 => leftBus,
                                  ip1 => rightBus,
                                  sel => controlWord(11),
```

```
                                op  => interPE );

  -- Connect the accumulator Mux's and register 's
  acc0Mux : mux2 generic map ( busWidth => dw )
                 port    map ( ip0 => interPE,
                               ip1 => shifterOut( dw - 1 downto 0 ),
                               sel => controlWord(12),
                               op  => accumulatorIn( dw - 1 downto 0 ) );

  -- Connect the accumulator Mux's and register 's
  acc1Mux : mux2 generic map ( busWidth => dw )
                 port    map ( ip0 => interPE,
                               ip1 => shifterOut( dw * 2 - 1 downto dw ),
                               sel => controlWord(13),
                               op  => accumulatorIn( dw * 2 - 1 downto dw ) );

  -- Connect the accumulator Mux's and register 's
  acc2Mux : mux2 generic map ( busWidth => dw )
                 port    map ( ip0 => interPE,
                               ip1 => shifterOut( dw * 3 - 1 downto dw * 2 ),
                               sel => controlWord(14),
                               op  => accumulatorIn( dw * 3 - 1 downto dw * 2 ) );

  -- Connect the accumulator reset signal
  nAccReset <= nReset and controlWord(15);

  -- In this case, controlWord(16) is not used.

end generate shiftBus1;

shiftBus3 : if sbw = dw * 3 generate

  -- Connect the accumulator Mux's and register 's
  accumulator0Mux : mux4 generic map ( busWidth => dw )
                         port    map ( ip0 => leftBus( dw - 1 downto 0 ),
                                       ip1 => rightBus( dw - 1 downto 0 ),
                                       ip2 => shifterOut( dw - 1 downto 0 ),
                                       ip3 => zeros( dw - 1 downto 0 ),
                                       sel => controlWord( 12 downto 11 ),
                                       op  => accumulatorIn( dw - 1 downto 0 )
                                     );

  -- Connect the accumulator Mux's and register 's
  accumulator1Mux : mux4 generic map ( busWidth => dw )
                         port    map ( ip0 => leftBus( dw * 2 - 1 downto dw ),
                                       ip1 => rightBus( dw * 2 - 1 downto dw ),
                                       ip2 => shifterOut( dw * 2 - 1 downto dw ),
                                       ip3 => zeros( dw - 1 downto 0 ),
                                       sel => controlWord( 14 downto 13 ),
                                       op  => accumulatorIn( dw * 2 - 1 downto dw )
                                     );

  -- Connect the accumulator Mux's and register 's
  accumulator2Mux : mux4 generic map ( busWidth => dw )
                         port    map ( ip0 => leftBus( dw * 3 - 1 downto dw * 2 ),
                                       ip1 => rightBus( dw * 3 - 1 downto dw * 2 ),
                                       ip2 => shifterOut( dw * 3 - 1 downto dw * 2 ),
                                       ip3 => zeros( dw * 3 - 1 downto dw * 2 ),
                                       sel => controlWord( 16 downto 15 ),
                                       op  => accumulatorIn( dw * 3 - 1 downto dw * 2 )
                                     );

  nAccReset <= nReset;

end generate shiftBus3;

accumulator0Enable          <= stackOutput and controlWord(17);
accumulator1Enable          <= stackOutput and controlWord(18);
accumulator2Enable          <= stackOutput and controlWord(19);

accumulator0 : reg generic map ( busWidth => dw )
                   port    map ( d        => accumulatorIn( dw - 1 downto 0 ),
                                 clock    => clock,
                                 nReset   => nAccReset,
                                 enable   => accumulator0Enable,
                                 q        => accumulatorOut( dw - 1 downto 0 ) );

accumulator1 : reg generic map ( busWidth => dw )
                   port    map ( d        => accumulatorIn( dw * 2 - 1 downto dw ),
                                 clock    => clock,
                                 nReset   => nAccReset,
                                 enable   => accumulator1Enable,
                                 q        => accumulatorOut( dw * 2 - 1 downto dw ) );

accumulator2 : reg generic map ( busWidth => dw )
                   port    map ( d        => accumulatorIn( dw * 3 - 1 downto dw * 2 ),
                                 clock    => clock,
                                 nReset   => nAccReset,
                                 enable   => accumulator2Enable,
                                 q        => accumulatorOut( dw * 3 - 1 downto dw * 2 ) );

-- Create the broadcast bus register
```

```
broadcastBusRegisterMux : mux4 generic map ( busWidth => dw )
                               port    map ( ip0      => ramDataA,
                                             ip1      => ramDataB,
                                             ip2      => broadcastBus,
                                             ip3      => zeros( dw - 1 downto 0 ),
                                             sel      => controlWord( 21 downto 20 ),
                                             op       => broadcastBusRegisterIn );

broadcastBusRegisterEnable <= stackOutput and controlWord( 22 );

broadcastBusRegister : reg generic map ( busWidth => dw )
                           port    map ( d        => broadcastBusRegisterIn,
                                         clock    => clock,
                                         nReset   => nReset,
                                         enable   => broadcastBusRegisterEnable,
                                         q        => broadcastBusRegisterOut );

-- Optionally include the arbiter circuitry
arbiterCircuitry : if ia = 1 generate

  arb : arbiter generic map ( busDepth => 4 )
                port    map ( broadcastBusRegister => broadcastBusRegisterOut,
                              broadcastBus         => broadcastBus,
                              clock                => clock,
                              enable               => controlWord( 28 ),
                              nReset               => nReset,
                              win                  => win(0) );

  -- Drive the broadcast bus
  broadcastBusOut <= broadcastBusRegisterOut when (win(0) = '1')
                                             else (others => '1');

end generate arbiterCircuitry;

noArbiter : if ia = 0 generate

  win(0)          <= '0';
  broadcastBusOut <= (others => '1');

end generate noArbiter;

-- Add the comparator
comparator : DW01_cmp6 generic map ( width => dw )
                       port    map ( A     => aluInputA,
                                     B     => aluInputB,
                                     TC    => logicOne,
                                     LT    => comparatorOut(LT),
                                     GT    => comparatorOut(GT),
                                     EQ    => comparatorOut(EQ),
                                     LE    => comparatorOut(LE),
                                     GE    => comparatorOut(GE),
                                     NE    => comparatorOut(NE) );

-- Add the stack
stackMux : mux8 generic map ( busWidth => 1 )
                   port    map ( ip0 => comparatorOut(LT downto LT),
                                 ip1 => comparatorOut(GT downto GT),
                                 ip2 => comparatorOut(EQ downto EQ),
                                 ip3 => comparatorOut(LE downto LE),
                                 ip4 => comparatorOut(GE downto GE),
                                 ip5 => comparatorOut(NE downto NE),
                                 ip6 => win,
                                 ip7 => zeros( 0 downto 0 ),
                                 sel => controlWord( 25 downto 23 ),
                                 op  => stackInput );

enableStack : stack generic map ( stackDepth => sd )
                    port    map ( dataIn  => stackInput(0),
                                  nReset  => nReset,
                                  clock   => clock,
                                  opMode  => controlWord( 27 downto 26 ),
                                  dataOut => stackOutput );

-- Generate enhanced routing data muxes
dataMuxes_er : if sbw = dw generate

  -- Connect BUS MUX's
  dataBusMux : mux4 generic map ( busWidth => dw )
                    port    map ( ip0 => accumulatorOut( dw - 1 downto 0 ),
                                  ip1 => accumulatorOut( dw * 2 - 1 downto dw ),
                                  ip2 => accumulatorOut( dw * 3 - 1 downto dw * 2 ),
                                  ip3 => inputAOut,
                                  sel => controlWord( 30 downto 29 ),
                                  op  => dataBus );

  busADriver : tri generic map ( busWidth => dw )
                   port    map ( input   => dataBus,
                                 enable  => controlWord( 31 ),
                                 output  => ramDataA );

  busBDriver : tri generic map ( busWidth => dw )
                   port    map ( input   => dataBus,
```

```
                                    enable   => controlWord( 32 ),
                                    output   => ramDataB );

end generate dataMuxes_er;

-- Generate non-enhanced routing data muxes
dataMuxes_ner : if sbw = dw * 3 generate

  -- Connect BUS MUX's
  dataBusAMux : mux4 generic map ( busWidth => dw )
                     port    map ( ip0 => accumulatorOut( dw - 1 downto 0 ),
                                   ip1 => accumulatorOut( dw * 2 - 1 downto dw ),
                                   ip2 => accumulatorOut( dw * 3 - 1 downto dw * 2 ),
                                   ip3 => zeros( 15 downto 0 ),
                                   sel => controlWord( 30 downto 29 ),
                                   op  => dataBusA );

  dataBusBMux : mux4 generic map ( busWidth => dw )
                     port    map ( ip0 => accumulatorOut( dw - 1 downto 0 ),
                                   ip1 => accumulatorOut( dw * 2 - 1 downto dw ),
                                   ip2 => accumulatorOut( dw * 3 - 1 downto dw * 2 ),
                                   ip3 => zeros( 15 downto 0 ),
                                   sel => controlWord( 43 downto 42 ),
                                   op  => dataBusB );

  busADriver : tri generic map ( busWidth => dw )
                   port    map ( input    => dataBusA,
                                 enable   => controlWord( 31 ),
                                 output   => ramDataA );

  busBDriver : tri generic map ( busWidth => dw )
                   port    map ( input    => dataBusB,
                                 enable   => controlWord( 32 ),
                                 output   => ramDataB );

end generate dataMuxes_ner;

driveShiftBus3 : if sbw = dw * 3 generate

  accDriveLeft3  : tri generic map ( busWidth => sbw )
                       port    map ( input    => accumulatorOut,
                                     enable   => controlWord( 33 ),
                                     output   => leftBus );

  accDriveRight3 : tri generic map ( busWidth => sbw )
                       port    map ( input    => accumulatorOut,
                                     enable   => controlWord( 34 ),
                                     output   => rightBus );

end generate driveShiftBus3;

driveShiftBus1 : if sbw = dw generate

  accDriveLeft1  : tri generic map ( busWidth => sbw )
                       port    map ( input    => dataBus,
                                     enable   => controlWord( 33 ),
                                     output   => leftBus );

  accDriveRight1 : tri generic map ( busWidth => sbw )
                       port    map ( input    => dataBus,
                                     enable   => controlWord( 34 ),
                                     output   => rightBus );

end generate driveShiftBus1;

-- Add the bidirectional drivers
leftDriver  : bidirectional generic map ( busWidth  => sbw )
                            port    map ( internal  => leftBus,
                                          direction => controlWord(35),
                                          external  => leftPE );

rightDriver : bidirectional generic map ( busWidth  => sbw )
                            port    map ( internal  => rightBus,
                                          direction => controlWord(36),
                                          external  => rightPE );

mapShiftReg : if isr = 1 generate

  shiftRegisterEnable <= stackOutput and controlWord( 39 );

  U1 : if sbw = dw * 3 generate

    -- Create the shift register
    shiftRegisterMux : mux4 generic map ( busWidth => sbw )
                            port    map ( ip0      => leftBus,
                                          ip1      => rightBus,
                                          ip2      => shifterOut,
                                          ip3      => zeros,
                                          sel      => controlWord( 38 downto 37 ),
                                          op       => shiftRegisterIn );

  end generate U1;
```

```
    U2 : If sbw = dw generate

      shiftRegMux : mux2 generic map ( busWidth => dw )
                    port     map ( ip0      => leftBus,
                                   ip1      => rightBus,
                                   sel      => controlWord( 37 ),
                                   op       => shiftRegisterIn );

    end generate U2;

    shiftRegister : reg generic map ( busWidth => sbw )
                    port     map ( d        => shiftRegisterIn,
                                   clock    => clock,
                                   nReset   => nReset,
                                   enable   => shiftRegisterEnable,
                                   q        => shiftRegisterOut );

    -- Drive the Left Bus
    leftDriverFromSR : tri generic map ( busWidth => sbw )
                           port     map ( input    => shiftRegisterOut,
                                          enable   => controlWord(40),
                                          output   => leftBus );

    -- Drive the Right Bus
    rightDriverFromSR : tri generic map ( busWidth => sbw )
                            port     map ( input    => shiftRegisterOut,
                                           enable   => controlWord(41),
                                           output   => rightBus );

  end generate mapShiftReg;

  ramEnable    <= stackOutput;

end architecture RTL;
```

# Appendix D

# Source Code for the Image Processing Algorithms

## D.1 Pentium 4 Source Code

Listing D.1: Pentium 4 Source Code

```
#include <iostream>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <ipp.h>
#include <ippcv.h>

#include "stopwatch.h"

#define THRESHOLD
#define SOBELDX
#define SOBELDY
#define DILATE
#define ERODE
#define BOXFILTER
#define MEDIANFILTER

#define LIMIT 100
static Ipp8u * imageBuffer;

using std::cerr;
using std::endl;
using std::cout;

void printUsage( char * programName )
{

  cerr << "Invalid argument:" << endl;
  cerr << programName << " -[txydebml] iimage.pgm " << endl;
  cerr << "-t = Threshold image" << endl;
  cerr << "-x = Sobel Dx image" << endl;
  cerr << "-y = Sobel Dy image" << endl;
  cerr << "-d = Dilate image" << endl;
  cerr << "-e = Erode image" << endl;
  cerr << "-b = Box Filter image" << endl;
  cerr << "-m = Median Filter image" << endl;
  cerr << endl;
  cerr << "-ln = Set Repition Limit" << endl;
  cerr << endl;
  cerr << "-iimage.pgm = Set Image data file (PGM format only!)" << endl;
  cerr << endl;

  exit(1);

} // printUsage

void findHeader( FILE * fp )
{

  char buffer[128];

  while (true)
  {
```

```
    if ( fgets( buffer, 128, fp ) == NULL )
    {
      cerr << "Not a valid PGM file, no header" << endl;
      abort();
    } // if
    if ( buffer[0] == '#' )
    {
      continue;
    } // if
    else if ( buffer[0] == 'P' )
    {
      if ( buffer[1] == '5' )
      {
        break;
      } // if
      else
      {
        cerr << "Invalid PGM valid, wrong header." << endl;
        abort();
      } // else
    } // else if
    else
    {
      cerr << "Not a valid PGM file, missing header" << endl;
      abort();
    } // else
  } // while
} // findHeader

IppiSize findSize( FILE * fp )
{
  IppiSize size;
  char buffer[128];
  while (true)
  {
    if ( fgets( buffer, 128, fp ) == NULL )
    {
      cerr << "Not a valid PGM file, no size" << endl;
      abort();
    } // if
    if ( buffer[0] == '#' )
    {
      continue;
    } // if
    else if ( buffer[0] >= '0' || buffer[0] <= '9' )
    {
      int num = sscanf( buffer, "%d %d\n", &size.width, &size.height );
      if ( num <= 1 )
      {
        cerr << "Invalid PGM file, corrupt size" << endl;
      } // if
      else
      {
        break;
      } // else
    } // else if
    else
    {
      cerr << "Not a valid PGM file, missing size" << endl;
      abort();
```

```
    } // else

  } // while

  return size;

} // findSize

int findMaxValue( FILE * fp )
{

  int  maxValue;
  char buffer[128];

  while (true)
  {

    if ( fgets( buffer, 128, fp ) == NULL )
    {

      cerr << "Not a valid PGM file, no maximum value" << endl;
      abort();

    } // if

    if ( buffer[0] == '#' )
    {

      continue;

    } // if
    else if ( buffer[0] >= '0' || buffer[0] <= '9' )
    {

      int num = sscanf( buffer, "%d\n", &maxValue );

      if ( num <= 0 )
      {

        cerr << "Invalid PGM file, corrupt maximum value" << endl;

      } // if
      else
      {

        break;

      } // else

    } // else if
    else
    {

      cerr << "Not a valid PGM file, missing maximum value" << endl;
      abort();

    } // else

  } // while

  return maxValue;

} // findMaxValue

IppiSize openImage( char * fileName )
{

  FILE *     fp;
  IppiSize   size;

  if ( (fp=fopen( fileName, "rb" )) == NULL )
  {

    cerr << "Could not open PGM file " << fileName << endl;
    abort();

  } // if

  findHeader( fp );
  size = findSize( fp );
  findMaxValue( fp );

  imageBuffer = new Ipp8u[ size.width * size.height ];
  assert( imageBuffer != NULL );

  for ( int i=0; i<size.width * size.height; i++ )
  {

    fread( &(imageBuffer[i]), sizeof( imageBuffer[i] ), 1, fp );

  } // for
```

```
  return size;

} // openImage

void dump( IppiSize size, const char * fileName, Ipp8u * image )
{

  FILE * fp;

  if ( ( fp = fopen( fileName, "wb" )) == NULL )
  {

    cerr << "Could not open file " << fileName << endl;
    abort();

  } // if

  fprintf( fp, "P5\n" );
  fprintf( fp, "%d %d\n", size.width, size.height );
  fprintf( fp, "255\n" );

  for ( int i=0; i<size.width * size.height; i++ )
  {

    fwrite( &image[i], sizeof( image[0] ), 1, fp );

  } // for

  fclose( fp );

} // dump

void dump( IppiSize size, const char * fileName, Ipp16s * image )
{

  FILE * fp;

  if ( ( fp = fopen( fileName, "wb" )) == NULL )
  {

    cerr << "Could not open file " << fileName << endl;
    abort();

  } // if

  fprintf( fp, "P5\n" );
  fprintf( fp, "%d %d\n", size.width, size.height );
  fprintf( fp, "65535\n" );

  for ( int i=0; i<size.width * size.height; i++ )
  {

    if ( image[i] < 0 )
    {

      image[i] = -image[i];

    } // if

    fwrite( &image[i], sizeof( image[0] ), 1, fp );

  } // for

  fclose( fp );

} // dump

int main( int argc, char ** argv )
{

  // Keeps track of the time for each algorithm
  double   count;

  // Image data Variables
  Ipp8u *  imageDestination8u;
  Ipp16s * imageDestination16s;
  IppiSize size;

  // Flags from command line
  bool threshold = false;
  bool dx        = false;
  bool dy        = false;
  bool dilate    = false;
  bool erode     = false;
  bool box       = false;
  bool median    = false;
  int  limit     = LIMIT;
  char * file    = NULL;

  // Argument index
  int arg = 1;
```

```
while ( arg < argc )
{
  if ( argv[arg][0] == '-' )
  {
    switch ( argv[arg][1] )
    {
      case 't' : threshold = true; break;
      case 'x' : dx        = true; break;
      case 'y' : dy        = true; break;
      case 'd' : dilate    = true; break;
      case 'e' : erode     = true; break;
      case 'b' : box       = true; break;
      case 'm' : median    = true; break;
      case 'a' :

        threshold = true;
        dx        = true;
        dy        = true;
        dilate    = true;
        erode     = true;
        box       = true;
        median    = true;

      break;

      case 'l' :

        limit = atoi( &argv[arg][2] );

      break;

      case 'i' :

        file = &argv[arg][2];

      break;

      default :

        printUsage( argv[0] );

      break;

    } // switch

  } // if
  else
  {

    printUsage( argv[0] );

  } // else

  // Process the next argument
  arg++;

} // while

if ( file == NULL )
{

  cerr << "Error!__Image_file_must_be_specified_(-i_option)" << endl;
  exit(1);

} // if

if ( limit <= 0 )
{

  cerr << "Error!__Limit_must_be_greater_than_0" << endl;
  exit(1);

} // if

size                = openImage( file );
imageDestination8u  = new Ipp8u[ size.width * size.height ];
imageDestination16s = new Ipp16s[ size.width * size.height ];

if (threshold)
{

  STARTCLOCK;

  for ( int i=0; i<limit; i++ )
  {

    ippiThreshold_LTValGTVal_8u_C1R( imageBuffer,
                                     size.width,
                                     imageDestination8u,
```

```
                                        size.width,
                                        size,
                                        (Ipp8u)127,
                                        (Ipp8u)0,
                                        (Ipp8u)127,
                                        (Ipp8u)255 );

  } // for

  STOPCLOCK(count)

  cout << "Threshold_Usage_time_=_" << count / (double)limit << "ms" << endl;

  dump( size, "sthreshold.pgm", imageDestination8u );

} // if

if (dx)
{

  STARTCLOCK;

  for ( int i=0; i<limit; i++ )
  {

    ippiSobel3x3_Dx_8u16s_C1R( imageBuffer,
                               size.width,
                               imageDestination16s,
                               size.width * 2,
                               size );

  } // for

  STOPCLOCK(count)

  cout << "Sobel_DX_Usage_time_=_" << count / (double)limit << "ms" << endl;

  dump( size, "sdx.pgm", imageDestination16s );

} // if

if (dy)
{

  STARTCLOCK;

  for ( int i=0; i<limit; i++ )
  {

    ippiSobel3x3_Dy_8u16s_C1R( imageBuffer,
                               size.width,
                               imageDestination16s,
                               size.width * 2,
                               0,
                               size );

  } // for

  STOPCLOCK(count)

  cout << "Sobel_DY_Usage_time_=_" << count / (double)limit << "ms" << endl;

  dump( size, "sdy.pgm", imageDestination16s );

} // if

if (dilate)
{

  Ipp8u * data = imageBuffer;
  Ipp8u * dst  = imageDestination8u;
  data += size.width + 1;
  dst  += size.width + 1;
  IppiSize roi = {size.width - 2, size.height - 2};

  // Zero the outlier pixels
  for ( int i=0; i<size.width; i++ )
  {

    imageDestination8u[ i ] = 0;
    imageDestination8u[ (size.height - 1) * size.width + i ] = 0;

  } // for

  for ( int i=0; i<size.height; i++ )
  {

    imageDestination8u[ size.width * i ] = 0;
    imageDestination8u[ (size.width - 1) * (i + 1 )] = 0;

  } // for
```

```
  STARTCLOCK;

  for ( int i=0; i<limit; i++ )
  {

    ippiDilate3x3_8u_C1R( data, size.width, dst, size.width, roi );

  } // for

  STOPCLOCK(count)

  cout << "Dilate_Usage_time_=_" << count / (double)limit << "ms" << endl;

  dump( size, "sdilate.pgm", imageDestination8u );

} // if

if (erode)
{

  Ipp8u * data = imageBuffer;
  Ipp8u * dst  = imageDestination8u;
  IppiSize roi = {size.width - 2, size.height - 2};

  data += size.width + 1;
  dst  += size.width + 1;

  // Zero the outlier pixels
  for ( int i=0; i<size.width; i++ )
  {

    imageDestination8u[ i ] = 0;
    imageDestination8u[ (size.height - 1) * size.width + i ] = 0;

  } // for

  for ( int i=0; i<size.height; i++ )
  {

    imageDestination8u[ size.width * i ] = 0;
    imageDestination8u[ (size.width - 1) * (i + 1 )] = 0;

  } // for

  STARTCLOCK;

  for ( int i=0; i<limit; i++ )
  {

    ippiErode3x3_8u_C1R( data, size.width, dst, size.width, roi );

  } // for

  STOPCLOCK(count)

  cout << "Erode_Usage_time_=_" << count / (double)limit << "ms" << endl;

  dump( size, "serode.pgm", imageDestination8u );

} // if

if (box)
{

  IppiSize maskSize = {3,3};
  IppiPoint anchor = {1,1};

  STARTCLOCK;

  for ( int i=0; i<limit; i++ )
  {

    ippiFilterBox_8u_C1R( imageBuffer,
                          size.width,
                          imageDestination8u,
                          size.width,
                          size,
                          maskSize,
                          anchor );

  } // for

  STOPCLOCK(count)

  cout << "Box_Filter_Usage_time_=_" << count / (double)limit << "ms" << endl;

  dump( size, "sbox.pgm", imageDestination8u );

} // if

if (median)
{
```

```
    IppiSize maskSize = {3,3};
    IppiPoint anchor = {1,1};

    STARTCLOCK;

    for ( int i=0; i<limit; i++ )
    {

      ippiFilterMedian_8u_C1R( imageBuffer,
                               size.width,
                               imageDestination8u,
                               size.width,
                               size,
                               maskSize,
                               anchor );

    } // for

    STOPCLOCK(count)

    cout << "Median_Filter_Usage_time_=_" << count / (double)limit << "ms" << endl;

    dump( size, "smedian.pgm", imageDestination8u );

  } // if

  // Release all the images
  delete[] imageBuffer;
  delete[] imageDestination8u;
  delete[] imageDestination16s;

  return 0;

} // main
```

# D.2 DSP-RAM Source Code

Listing D.2: Image Class Definition

```
// $Id: image.h,v 1.4 2002/08/07 21:11:42 dillen Exp $
//
// $Log: image.h,v $
// Revision 1.4  2002/08/07 21:11:42  dillen
// Added temporary storage in both banks
//
// Revision 1.3  2002/08/01 16:05:56  dillen
// Added all the imaging functions
//
// Revision 1.2  2002/07/29 20:34:10  dillen
// Added box filter, filter3x3, dilate and erode operators
//
// Revision 1.1  2002/07/23 20:57:20  dillen
// Initial revision
//
// Revision 1.1  2002/07/22 22:09:47  dillen
// Initial revision
//

#include <fstream>
#include <dspramClass.h>
#include <integerClasses.h>

class image
{

  public:

    image( char * fileName,
           int    numberOfPEs = NUM_PEs,
           int    ramSize = RAM_BANK_SIZE );

    virtual ~image( void );

    // Debug Functions
    void copy( void );
    void dump( char * fileName );
    void dumpA( char * fileName );
    int  getClockCycles( void );

    // Application functions
    void threshold( Int16 minimum, Int16 maximum, Int16 threshold );
    void filter3x3( Int16 coefficients[9] );
    void dilate( void );
    void erode( void );
    void box( void );
    void medianFilter( void );
```

```
private:

  // Private data members
  DSPRAMClass dspram;
  int         imageAddressA;
  int         imageAddressB;
  int         oneA;            // Is this necessary ???
  int         oneB;
  int         PENumberB;
  int         pixelNeighbourhoodA;
  int         boxFilterUpperA;
  int         boxFilterLowerA;
  int         comparisonPixelB;
  int         sortingArrayB;
  int         temporaryStorageA;
  int         temporaryStorageB;

  int         width;
  int         height;
  int         columnImageSize;
  int         numberOfPEs;

  // Private functions
  void load( char * fileName );
  void parsePGMHeader( std::ifstream & file );
  void parsePGMSize( std::ifstream & file );
  void parsePGMMaxValue( std::ifstream & file );

  // utility functions
  void collectNeighbours( int index );
  void compareMax( int addressA, int byte );
  void compareMin( int addressA, int byte );
  void sortNeighbourhood( void );
  void dumpNeighbourhood( int pe );

  void parsePGMHeader( FILE * file );
  void parsePGMSize( FILE * file );
  void parsePGMMaxValue( FILE * file );

};
```

## Listing D.3: Image Class Implementation

```
// $Id: image.cc,v 1.4 2002/08/06 16:53:36 dillen Exp dillen $
//
// $Log: image.cc,v $
// Revision 1.4  2002/08/06 16:53:36  dillen
// Added implementation for a 16-bit shift bus
//
// Revision 1.3  2002/08/01 16:05:56  dillen
// Added all the imaging functions
//
// Revision 1.2  2002/07/29 20:34:10  dillen
// Added box filter, filter3x3, dilate and erode operators
//
// Revision 1.1  2002/07/23 20:57:20  dillen
// Initial revision
//

#include <fstream>
#include <sstream>
#include <dspramClass.h>
#include <integerClasses.h>
#include "image.h"

#ifndef CPP_WORKING
#include <stdlib.h>
#endif

image::image( char * fileName, int numberOfPEs, int ramBankSize )

: dspram( numberOfPEs, ramBankSize )

{

  int i;
  int pe;

  // Load the file
  load( fileName );

  // Allocate a constant of one in each bank
  oneA       = dspram.intAlloc( BANK_A );
  oneB       = dspram.intAlloc( BANK_B );

  // Allocate the PE #
  PENumberB = dspram.intAlloc( BANK_B );

  // Write a one in each bank serially
```

```
for ( i = 0; i < numberOfPEs; i++ )
{

  dspram.startInstructionDefn();

    dspram.write( 1, i, BANK_A, oneA );
    dspram.write( 1, i, BANK_B, oneB );

  dspram.endInstructionDefn();

} // for

// Shift in the PE Numbers
for ( pe = dspram.getNumberOfPEs(); pe >= 0; pe-- )
{

  dspram.startInstructionDefn();

    dspram.rightBusFromAcc();
    dspram.injectIntoLeft( pe - 1 );
    dspram.accLoadFromLeft();

  dspram.endInstructionDefn();

} // for

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_B, PENumberB, ACC_0 );

dspram.endInstructionDefn();

// Allocate the pixel neighbourhood window
pixelNeighbourhoodA = dspram.intVectorAlloc( BANK_A, 9 );

// Allocate box filter storage
boxFilterUpperA = dspram.intAlloc( BANK_A );
boxFilterLowerA = dspram.intAlloc( BANK_A );

// Allocate comparison pixel storage
comparisonPixelB = dspram.intAlloc( BANK_B );

// Allocate sorting storage
sortingArrayB = dspram.intVectorAlloc( BANK_B, 9 );

// Allocate temporary storage
temporaryStorageA = dspram.intAlloc( BANK_A );
temporaryStorageB = dspram.intAlloc( BANK_B );

} // image constructor

image::~image()
{
} // image destructor

int image::getClockCycles( void )
{

  return dspram.getCycleCount();

} // getClockCycles

void image::copy( void )
{

  int i;

  for ( i = 0; i < columnImageSize; i++ )
  {

    dspram.startInstructionDefn();

      dspram.enableByteSelect(0);
      dspram.multAxB( imageAddressA + i, oneB );
      dspram.disableAdder();
      dspram.setShifter( noShift );

    dspram.endInstructionDefn();
    // M0 = lower

    dspram.startInstructionDefn();

      dspram.enableByteSelect(1);
      dspram.multAxB( imageAddressA + i, oneB );
      dspram.disableAdder();
      dspram.setShifter( noShift );

    dspram.endInstructionDefn();
    // M0 = upper, M1 = lower

    dspram.startInstructionDefn();
    dspram.endInstructionDefn();
```

```
    // M0 = X, M1 = upper, M2 = lower

    dspram.startInstructionDefn();

      dspram.acc0LoadShifter();

    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = upper, Acc0 = lower

    dspram.startInstructionDefn();

      dspram.acc0LoadShifter();
      dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i, ACC_0 );

    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = upper, Acc0 = lower

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i + 1, ACC_0 );

    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = X, Acc0 = upper

  } // for

} // copy

void image::dump( char * fileName )
{

#ifdef CPP_WORKING
  int    i;
  int    pe;
  char   buffer;
  Int16  data;

  std::ofstream file( fileName, std::ios::out | std::ios::binary );

  file << "P5" << std::endl;
//  file << "# Created by DSPRAMImage class from BANK_B" << std::endl;
  file << width << " " << height << std::endl;
  file << 255 << std::endl;

  for ( i = 0; i < columnImageSize; i++ )
  {

    for ( pe = 0; pe < numberOfPEs; pe++ )
    {

//      dspram.startInstructionDefn();

      data = dspram.read( pe, BANK_B, imageAddressB + 2 * i );

//      dspram.endInstructionDefn();

      if ( data < (Int16)0 )
      {

        buffer = (char)-data;

      } // if
      else
      {

        buffer = (char)data;

      } // else

      file << buffer;

//      dspram.startInstructionDefn();

      data = dspram.read( pe, BANK_B, imageAddressB + 2 * i + 1 );

//      dspram.endInstructionDefn();

      if ( data < (Int16)0 )
      {

        buffer = (char)-data;

      } // if
      else
      {

        buffer = (char)data;

      } // else

      file << buffer;
```

```
    } // for

  } // for

#else

  FILE * ofp;

  if ( (ofp=fopen( fileName, "wb" )) == NULL )
  {

    std::cerr << "Could not open file " << fileName << std::endl;
    abort();

  } // if

  fprintf( ofp, "P5\n# Created by DSPRAMImage class from BANK B\n" );
//  fprintf( ofp, "P5\n" );
  fprintf( ofp, "%d %d\n%d\n", width, height, 255 );

  unsigned char buffer[numberOfPEs * 2];

  for ( int row = 0; row < columnImageSize; row++ )
  {

    for ( int pe=0; pe<numberOfPEs; pe++ )
    {

      Int16 value = dspram.read( pe, BANK_B, imageAddressB + 2 * row + 0 );

      if ( value < (Int16)0 )
      {

        buffer[pe * 2 + 0] = -value;

      } // if
      else
      {

        buffer[pe * 2 + 0] = value;

      } // end if

      value = dspram.read( pe, BANK_B, imageAddressB + 2 * row + 1 );

      if ( value < (Int16)0 )
      {

        buffer[pe * 2 + 1] = -value;

      } // if
      else
      {

        buffer[pe * 2 + 1] = value;

      } // end if

    } // for

    fwrite( buffer, sizeof( buffer[0] ), numberOfPEs * 2, ofp );

  } // for

  fclose( ofp );

#endif

} // image dump

void image::dumpA( char * fileName )
{

  int i;
  int pe;
  Int16 data;
  short shortData;

  std::ofstream file( fileName, std::ios::out | std::ios::binary );

  file << "P5" << std::endl << "# Created by image class from BANK A" << std::endl;
  file << width << " " << height << " " << std::endl;
  file << 255 << std::endl;

  for ( i = 0; i < columnImageSize; i++ )
  {

    for ( pe = 0; pe < numberOfPEs; pe++ )
    {

      data = dspram.read( pe, BANK_A, imageAddressA + i );
      shortData = (short)data & 0x00FF;
```

```
      file << (char)shortData;
      shortData = ((short)data & 0xFF00) >> 8;
      file << (char)shortData;

    } // for

  } // for

} // image dump

void image::collectNeighbours( int index )
{

#ifdef SHIFTBUS_16

#ifdef ENHANCED_ROUTING

  int offset;
  Int16 busValue;

  offset = width / (numberOfPEs * 2);

  // Step 1: Load accumulator with the three pixel locations
  // Step 2: Shift accumulator right.
  // Step 3: Store three pixels into memory.
  // Step 4: Shift accumulator left.
  // Step 5: Store three pixels into memory.
  // Step 6: If there is more columns then PEs then
  //         send PE 0 data to PE N-1 and vice versa

#if DEBUG
  std::cout << "Starting collection with enhanced routing..." << std::endl;
#endif

  // Store pixel into the multiplier register
  dspram.startInstructionDefn();

    dspram.disableByteSelect();
    dspram.saveInputA( BANK_A, imageAddressA + index - offset );

  dspram.endInstructionDefn( DEBUG );

  // Store the pixel value as well as shift it left
  dspram.startInstructionDefn();

    dspram.injectIntoRight( 0 );
    dspram.leftBusFromInputA();
    dspram.memLoadInputA( BANK_A, pixelNeighbourhoodA + 1 );
    dspram.accLoadFromRight( WORD_0 );

  dspram.endInstructionDefn( DEBUG );
  // ACC0 = UR

  // Store the pixel value as well as shift it right
  dspram.startInstructionDefn();

    dspram.injectIntoLeft( 0 );
    dspram.rightBusFromInputA();
    dspram.accLoadFromLeft( WORD_1 );

  dspram.endInstructionDefn( DEBUG );
  // ACC1 = UL, ACC0 = UR

  // Store the pixel value
  dspram.startInstructionDefn();

    dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 0, ACC_1 );

  dspram.endInstructionDefn( DEBUG );
  // ACC1 = UL, ACC0 = UR

  // Store the pixel value
  dspram.startInstructionDefn();

    dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 2, ACC_0 );

  dspram.endInstructionDefn( DEBUG );
  // ACC1 = UL, ACC0 = UR

  // Store middle pixel into the multiplier register
  dspram.startInstructionDefn();

    dspram.saveInputA( BANK_A, imageAddressA + index );

  dspram.endInstructionDefn( DEBUG );

  // Store the pixel value as well as shift it left
  dspram.startInstructionDefn();

    dspram.injectIntoRight( 0 );
    dspram.leftBusFromInputA();
    dspram.memLoadInputA( BANK_A, pixelNeighbourhoodA + 4 );
```

```
  dspram.accLoadFromRight( WORD_0 );

dspram.endInstructionDefn( DEBUG );
// ACC0 = UR

// Store the pixel value as well as shift it right
dspram.startInstructionDefn();

  dspram.injectIntoLeft( 0 );
  dspram.rightBusFromInputA();
  dspram.accLoadFromLeft( WORD_1 );

dspram.endInstructionDefn( DEBUG );
// ACC1 = UL, ACC0 = UR

// Store the pixel value
dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 3, ACC_1 );

dspram.endInstructionDefn( DEBUG );
// ACC1 = UL, ACC0 = UR

// Store the pixel value
dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 5, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// ACC1 = UL, ACC0 = UR

// Store middle pixel into the multiplier register
dspram.startInstructionDefn();

  dspram.saveInputA( BANK_A, imageAddressA + index + offset );

dspram.endInstructionDefn( DEBUG );

// Store the pixel value as well as shift it left
dspram.startInstructionDefn();

  dspram.injectIntoRight( 0 );
  dspram.leftBusFromInputA();
  dspram.memLoadInputA( BANK_A, pixelNeighbourhoodA + 7 );
  dspram.accLoadFromRight( WORD_0 );

dspram.endInstructionDefn( DEBUG );
// ACC0 = UR

// Store the pixel value as well as shift it right
dspram.startInstructionDefn();

  dspram.injectIntoLeft( 0 );
  dspram.rightBusFromInputA();
  dspram.accLoadFromLeft( WORD_1 );

dspram.endInstructionDefn( DEBUG );
// ACC1 = UL, ACC0 = UR

// Store the pixel value
dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 6, ACC_1 );

dspram.endInstructionDefn( DEBUG );
// ACC1 = UL, ACC0 = UR

// Store the pixel value
dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 8, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// ACC1 = UL, ACC0 = UR

// Load accumulator with last PE's neighbours
// (PE(0) sends to PE(N-1)
if ( ( index + 1 ) % offset )
{

  // Load accumulator with last PE's neighbours
  dspram.startInstructionDefn();

    dspram.saveInputA( BANK_A, imageAddressA + index + 1 - offset );

  dspram.endInstructionDefn( DEBUG );
  // MA = UR

  // Shift left
  dspram.startInstructionDefn();

    dspram.injectIntoRight(0);
```

```
    dspram.leftBusFromInputA();
    busValue = dspram.extractFromLeft();
    dspram.saveInputA( BANK_A, imageAddressA + index + 1 );

  dspram.endInstructionDefn( DEBUG );
  // Bus = UR, MA = MR

  dspram.startInstructionDefn();

    dspram.injectIntoRight(busValue);
    dspram.leftBusFromInputA();
    busValue = dspram.extractFromLeft();
    dspram.saveInputA( BANK_A, imageAddressA + index + 1 + offset );
    dspram.accLoadFromRight( WORD_0 );

  dspram.endInstructionDefn( DEBUG );
  // Acc0 = UR, Bus = MR, MA = LR

  dspram.startInstructionDefn();

    dspram.injectIntoRight(busValue);
    dspram.leftBusFromInputA();
    busValue = dspram.extractFromLeft();
    dspram.accLoadFromRight( WORD_1 );
    dspram.injectOntoBroadcastBus( numberOfPEs - 1 );
    dspram.regLoadBroadcastBus();

  dspram.endInstructionDefn( DEBUG );
  // Acc1 = MR, Acc0 = UR, Bus = LR, R = N - 1

  dspram.startInstructionDefn();

    dspram.injectIntoRight( busValue );
    dspram.leftBusFromInputA();
    dspram.accLoadFromRight( WORD_2 );
    dspram.pushEnableBeqR( PENumberB );

  dspram.endInstructionDefn( DEBUG );
  // IF PE == PE(N-1), Acc2 = LR, Acc1 = MR, Acc0 = UR

    // Shift the neighbours into the accumulator
    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 2, ACC_0 );

    dspram.endInstructionDefn( DEBUG );

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 5, ACC_1 );

    dspram.endInstructionDefn( DEBUG );
    // Acc0 = UM

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 8, ACC_2 );
      dspram.popEnable();

    dspram.endInstructionDefn( DEBUG );

  // END IF

} // if

// Check to see if PE 0 requires pixels to be shifted in.
If ( index % offset )
{

  // Load accumulator with first PE's neighbours
  dspram.startInstructionDefn();

    dspram.disableByteSelect();
    dspram.saveInputA( BANK_A, imageAddressA + index - 1 - offset );

  dspram.endInstructionDefn( DEBUG );
  // MA = UL

  // Shift left
  dspram.startInstructionDefn();

    dspram.injectIntoLeft(0);
    dspram.rightBusFromInputA();
    busValue = dspram.extractFromRight();
    dspram.saveInputA( BANK_A, imageAddressA + index - 1 );

  dspram.endInstructionDefn( DEBUG );
  // MA = ML, BV = UL

  dspram.startInstructionDefn();

    dspram.injectIntoLeft(busValue);
```

```
      dspram.rightBusFromInputA();
      busValue = dspram.extractFromRight();
      dspram.saveInputA( BANK_A, imageAddressA + index - 1 + offset );
      dspram.accLoadFromLeft( WORD_0 );

    dspram.endInstructionDefn( DEBUG );
    // Acc0 = UL, BV = ML, MA = LL

    dspram.startInstructionDefn();

      dspram.injectIntoLeft(busValue);
      dspram.rightBusFromInputA();
      busValue = dspram.extractFromRight();
      dspram.accLoadFromLeft( WORD_1 );
      dspram.injectOntoBroadcastBus ( 0 );
      dspram.regLoadBroadcastBus();

    dspram.endInstructionDefn( DEBUG );
    // Acc1 = ML, Acc0 = UL, BV = LL, R = 0

    dspram.startInstructionDefn();

      dspram.injectIntoLeft( busValue );
      dspram.rightBusFromInputA();
      dspram.accLoadFromLeft( WORD_2 );
      dspram.pushEnableBeqR( PENumberB );

    dspram.endInstructionDefn( DEBUG );
    // IF PE == PE(0), Acc2 = LL, Acc1 = ML, Acc0 = UL, R = 0

      // Shift the neighbours into the accumulator
      dspram.startInstructionDefn();

        dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 0, ACC_0 );

      dspram.endInstructionDefn( DEBUG );

      dspram.startInstructionDefn();

        dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 3, ACC_1 );

      dspram.endInstructionDefn( DEBUG );

      dspram.startInstructionDefn();

        dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 6, ACC_2 );
        dspram.popEnable();

      dspram.endInstructionDefn( DEBUG );

    // END IF

  } // if

#else

  int offset;
  Int16 busValue[3];

  offset = width / (numberOfPEs * 2);

  // Step 1: Load accumulator with the three pixel locations
  // Step 2: Shift accumulator right.
  // Step 3: Store three pixels into memory.
  // Step 4: Shift accumulator left.
  // Step 5: Store three pixels into memory.
  // Step 6: If there is more columns then PEs then
  //         send PE 0 data to PE N-1 and vice versa.

#if DEBUG
  std::cout << "Starting_collection..." << std::endl;
#endif

  // Load the 3 pixel values into the accumulator
  dspram.startInstructionDefn();

    dspram.multAxB( imageAddressA + index - offset, oneB );
    dspram.disableAdder();
    dspram.setShifter( noShift );

  dspram.endInstructionDefn( DEBUG );
  // M0 = UM

  dspram.startInstructionDefn();
  dspram.endInstructionDefn( DEBUG );
  // M0 = X, M1 = UM

  dspram.startInstructionDefn();
  dspram.endInstructionDefn( DEBUG );
  // M0 = X, M1 = X, M2 = UM

  dspram.startInstructionDefn();
```

```
  dspram.multAxB( imageAddressA + index, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn( DEBUG );
// M0 = MM, M1 = X, M2 = X, Acc = UM

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 1, ACC_0 );
  dspram.injectIntoRight( 0 );          // Inject 0 pixels into outliers
  dspram.leftBusFromAcc( WORD_0 );
  dspram.accLoadFromRight( WORD_1 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = MM, M2 = X, Acc0 = UM, Acc1 = UR

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 2, ACC_1 );
  dspram.injectIntoLeft( 0 );          // Inject 0 pixels into outliers
  dspram.rightBusFromAcc( WORD_0 );
  dspram.accLoadFromLeft( WORD_2 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = MM, Acc0 = UM, Acc1 = UR, Acc2 = UL

dspram.startInstructionDefn();

  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 0, ACC_2 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Acc0 = MM, Acc1 = X, Acc2 = X

dspram.startInstructionDefn();

  dspram.multAxB( imageAddressA + index + offset, oneB );
  dspram.injectIntoRight( 0 );          // Inject 0 pixels into outliers
  dspram.leftBusFromAcc( WORD_0 );
  dspram.accLoadFromRight( WORD_1 );

dspram.endInstructionDefn( DEBUG );
// M0 = LM, M1 = X, M2 = X, Acc0 = MM, Acc1 = MR, Acc2 = X

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 4, ACC_0 );
  dspram.injectIntoLeft( 0 );          // Inject 0 pixels into outliers
  dspram.rightBusFromAcc( WORD_0 );
  dspram.accLoadFromLeft( WORD_2 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = LM, M2 = X, Acc0 = MM, Acc1 = MR, Acc2 = ML

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 3, ACC_2 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = LM, Acc0 = MM, Acc1 = MR, Acc2 = ML

dspram.startInstructionDefn();

  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 5, ACC_1 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Acc0 = LM, Acc1 = X, Acc2 = X

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 7, ACC_0 );
  dspram.injectIntoRight( 0 );
  dspram.leftBusFromAcc( WORD_0 );
  dspram.accLoadFromRight( WORD_1 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Acc0 = LM, Acc1 = LR, Acc2 = X

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 8, ACC_1 );
  dspram.injectIntoLeft( 0 );
  dspram.rightBusFromAcc( WORD_0 );
  dspram.accLoadFromLeft( WORD_2 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Acc0 = LM, Acc1 = LR, Acc2 = LL

dspram.startInstructionDefn();
```

```
    dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 6, ACC_2 );

  dspram.endInstructionDefn( DEBUG );
  // M0 = X, M1 = X, M2 = X, Acc0 = LM, Acc1 = LR, Acc2 = LL

  // Load accumulator with last PE's neighbours
  // (PE(0) sends to PE(N-1)
  if ( ( index + 1 ) % offset )
  {

    // Load accumulator with last PE's neighbours
    dspram.startInstructionDefn();

      dspram.multAxB( imageAddressA + index + 1 - offset, oneB );

    dspram.endInstructionDefn();
    // M0 = UM

    dspram.startInstructionDefn();

      dspram.multAxB( imageAddressA + index + 1, oneB );

    dspram.endInstructionDefn();
    // M0 = MM, M1 = UM

    dspram.startInstructionDefn();

      dspram.multAxB( imageAddressA + index + 1 + offset, oneB );

    dspram.endInstructionDefn();
    // M0 = LM, M1 = MM, M2 = UM

    dspram.startInstructionDefn();

      dspram.accLoadShifter();

    dspram.endInstructionDefn();
    // M0 = X, M1 = LM, M2 = MM, Acc0 = UM

    dspram.startInstructionDefn();

      dspram.injectIntoRight( 0 );
      dspram.leftBusFromAcc( WORD_0 );
      dspram.accLoadShifter();
      busValue[0] = dspram.extractFromLeft();

    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = LM, Acc0 = MM, busValue[0] = UM

    dspram.startInstructionDefn();

      dspram.injectIntoRight( 0 );
      dspram.leftBusFromAcc( WORD_0 );
      dspram.accLoadShifter();
      busValue[1] = dspram.extractFromLeft();

    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = X, Acc0 = LM, busValue[1] = MM, busValue[0] = UM

    dspram.startInstructionDefn();

      dspram.injectIntoRight( 0 );
      dspram.leftBusFromAcc( WORD_0 );
      busValue[2] = dspram.extractFromLeft();
      dspram.injectOntoBroadcastBus( numberOfPEs - 1 );
      dspram.regLoadBroadcastBus();

    dspram.endInstructionDefn();
    // busValue[2] = LM, busValue[1] = MM, busValue[0] = UM

#if DEBUG

  std::cout << std::hex << "Bus_Values_=_" << std::endl
            << busValue[0] << std::endl
            << busValue[1] << std::endl
            << busValue[2] << std::endl
            << std::dec;

#endif
    // Load the neighbours into memory
    dspram.startInstructionDefn();

      dspram.pushEnableBeqR( PENumberB );

    dspram.endInstructionDefn( DEBUG );
    // IF PENumberB == numberOfPEs - 1

      // Shift the neighbours into the accumulator
      dspram.startInstructionDefn();

        dspram.injectIntoRight( busValue[0] );
        dspram.leftBusFromAcc( WORD_0 );
```

```
      dspram.accLoadFromRight( WORD_0 );

    dspram.endInstructionDefn( DEBUG );
    // Acc0 = UM

    dspram.startInstructionDefn();

      dspram.injectIntoRight( busValue[1] );
      dspram.leftBusFromAcc( WORD_0 );
      dspram.accLoadFromRight( WORD_0 );
      dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 2, ACC_0 );

    dspram.endInstructionDefn( DEBUG );
    // Acc0 = MM

    dspram.startInstructionDefn();

      dspram.injectIntoRight( busValue[2] );
      dspram.leftBusFromAcc( WORD_0 );
      dspram.accLoadFromRight( WORD_0 );
      dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 5, ACC_0 );

    dspram.endInstructionDefn( DEBUG );
    // Acc0 = LM

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 8, ACC_0 );
      dspram.popEnable();

    dspram.endInstructionDefn( DEBUG );

  // END IF

} // if

// Check to see if PE 0 requires pixels to be shifted in.
if ( index % offset )
{

  // Load accumulator with last PE's neighbours
  dspram.startInstructionDefn();

    dspram.multAxB( imageAddressA + index - 1 - offset, oneB );

  dspram.endInstructionDefn();
  // M0 = UM

  dspram.startInstructionDefn();

    dspram.multAxB( imageAddressA + index - 1, oneB );

  dspram.endInstructionDefn();
  // M0 = MM, M1 = UM

  dspram.startInstructionDefn();

    dspram.multAxB( imageAddressA + index - 1 + offset, oneB );

  dspram.endInstructionDefn();
  // M0 = LM, M1 = MM, M2 = UM

  dspram.startInstructionDefn();

    dspram.accLoadShifter();

  dspram.endInstructionDefn();
  // M0 = X, M1 = LM, M2 = MM, Acc0 = UM

  dspram.startInstructionDefn();

    dspram.injectIntoLeft( 0 );
    dspram.rightBusFromAcc( WORD_0 );
    dspram.accLoadShifter();
    busValue[0] = dspram.extractFromRight();

  dspram.endInstructionDefn();
  // M0 = X, M1 = X, M2 = LM, Acc0 = MM, busValue[0] = UM

  dspram.startInstructionDefn();

    dspram.injectIntoLeft( 0 );
    dspram.rightBusFromAcc( WORD_0 );
    dspram.accLoadShifter();
    busValue[1] = dspram.extractFromRight();

  dspram.endInstructionDefn();
  // M0 = X, M1 = X, M2 = X, Acc0 = LM, busValue[1] = MM, busValue[0] = UM

  dspram.startInstructionDefn();

    dspram.injectIntoLeft( 0 );
```

```
      dspram.rightBusFromAcc( WORD_0 );
      busValue[2] = dspram.extractFromRight();
      dspram.injectOntoBroadcastBus ( 0 );
      dspram.regLoadBroadcastBus();

    dspram.endInstructionDefn();
    // busValue[2] = LM, busValue[1] = MM, busValue[0] = UM

    // Load the neighbours into memory
    dspram.startInstructionDefn();

      dspram.pushEnableBeqR( PENumberB );

    dspram.endInstructionDefn();
    // IF PENumberB == 0

      // Shift the neighbours into the accumulator
      dspram.startInstructionDefn();

        dspram.injectIntoLeft( busValue[0] );
        dspram.rightBusFromAcc( WORD_0 );
        dspram.accLoadFromLeft( WORD_0 );

      dspram.endInstructionDefn();
      // Acc0 = UM

      dspram.startInstructionDefn();

        dspram.injectIntoLeft( busValue[1] );
        dspram.rightBusFromAcc( WORD_0 );
        dspram.accLoadFromLeft( WORD_0 );
        dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 0, ACC_0 );

      dspram.endInstructionDefn();
      // Acc0 = MM

      dspram.startInstructionDefn();

        dspram.injectIntoLeft( busValue[2] );
        dspram.rightBusFromAcc( WORD_0 );
        dspram.accLoadFromLeft( WORD_0 );
        dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 3, ACC_0 );

      dspram.endInstructionDefn();
      // Acc0 = LM

      dspram.startInstructionDefn();

        dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 6, ACC_0 );
        dspram.popEnable();

      dspram.endInstructionDefn();

    // END IF

  } // if

#endif // ENHANCED_ROUTING

#else // 48-Bit Shiftbus

  int offset;
  Int48 busValue;

  offset = width / (numberOfPEs * 2);

  // Step 1: Load accumulator with the three pixel locations
  // Step 2: Shift accumulator right.
  // Step 3: Store three pixels into memory.
  // Step 4: Shift accumulator left.
  // Step 5: Store three pixels into memory.
  // Step 6: If there is more columns then PEs then
  //         send PE 0 data to PE N-1 and vice versa.

  // Load the 3 pixel values into the accumulator
  dspram.startInstructionDefn();

    dspram.multAxB( imageAddressA + index - offset, oneB );
    dspram.disableAdder();

  dspram.endInstructionDefn();
  // M0 = Upper

  dspram.startInstructionDefn();

    dspram.multAxB( imageAddressA + index, oneB );

  dspram.endInstructionDefn();
  // M0 = Mid, M1 = Upper

  dspram.startInstructionDefn();
```

```
  dspram.multAxB( imageAddressA + index + offset , oneB );

dspram.endInstructionDefn();
// M0 = Lower, M1 = Mid, M2 = Upper

dspram.startInstructionDefn();

  dspram.acc0LoadShifter();
  dspram.setShifter( LSL16 );

dspram.endInstructionDefn();
// M0 = X, M1 = Lower, M2 = Mid, Acc0 = Upper

dspram.startInstructionDefn();

  dspram.acc1LoadShifter();
  dspram.setShifter( LSL32 );
  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 1, ACC_0 );

dspram.endInstructionDefn();
// M0 = X, M1 = X, M2 = Lower, Acc1 = Mid, Acc0 = Upper

dspram.startInstructionDefn();

  dspram.acc2LoadShifter();
  dspram.setShifter( noShift );
  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 4, ACC_1 );

dspram.endInstructionDefn();
// M0 = X, M1 = X, M2 = X, Acc2 = Lower, Acc1 = Mid, Acc0 = Upper

// Shift pixel values left and load right neighbours
dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 7, ACC_2 );
  dspram.injectIntoRight( Int48(0) );
  dspram.leftBusFromAcc();
  dspram.accLoadFromRight();

dspram.endInstructionDefn();
// M0 = X, M1 = X, M2 = X, Accumulator = right neighbours

// Save right neighbours into memory
dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 2, ACC_0 );

dspram.endInstructionDefn();
// M0 = X, M1 = X, M2 = X, Accumulator = right neighbours

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 5, ACC_1 );

dspram.endInstructionDefn();
// M0 = X, M1 = X, M2 = X, Accumulator = right neighbours

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 8, ACC_2 );

dspram.endInstructionDefn();
// M0 = X, M1 = X, M2 = X, Accumulator = right neighbours

// Reload the original 3 image pixels
dspram.startInstructionDefn();

  dspram.multAxB( imageAddressA + index - offset , oneB );

dspram.endInstructionDefn();
// M0 = Upper

dspram.startInstructionDefn();

  dspram.multAxB( imageAddressA + index , oneB );

dspram.endInstructionDefn();
// M0 = Mid, M1 = Upper

dspram.startInstructionDefn();

  dspram.multAxB( imageAddressA + index + offset , oneB );

dspram.endInstructionDefn();
// M0 = Lower, M1 = Mid, m2 = Upper

dspram.startInstructionDefn();

  dspram.acc0LoadShifter();
  dspram.setShifter( LSL16 );

dspram.endInstructionDefn();
```

```
// M0 = X, M1 = Lower, M2 = Mid, ACC0 = Upper

dspram.startInstructionDefn();

  dspram.acc1LoadShifter();
  dspram.setShifter( LSL32 );

dspram.endInstructionDefn();
// M0 = X, M1 = X, M2 = Lower, ACC1 = Mid, ACC0 = Upper

dspram.startInstructionDefn();

  dspram.acc2LoadShifter();
  dspram.setShifter( noShift );

dspram.endInstructionDefn();
// M0 = X, M1 = X, M2 = X, ACC2 = Lower, ACC1 = Mid, ACC0 = Upper

// Shift pixels right and store left neighbours
dspram.startInstructionDefn();

  dspram.injectIntoLeft( Int48(0) );
  dspram.rightBusFromAcc();
  dspram.accLoadFromLeft();

dspram.endInstructionDefn();

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 0, ACC_0 );

dspram.endInstructionDefn();

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 3, ACC_1 );

dspram.endInstructionDefn();

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 6, ACC_2 );
  dspram.injectOntoBroadcastBus ( numberOfPEs - 1 );
  dspram.regLoadBroadcastBus();

dspram.endInstructionDefn();

// Load accumulator with last PE's neighbours
If ( ( index + 1 ) % offset )
{

  // Load accumulator with last PE's neighbours
  dspram.startInstructionDefn();

    dspram.multAxB( imageAddressA + index + 1 - offset, oneB );

  dspram.endInstructionDefn();
  // M0 = Upper

  dspram.startInstructionDefn();

    dspram.multAxB( imageAddressA + index + 1, oneB );

  dspram.endInstructionDefn();
  // M0 = Mid, M1 = Upper

  dspram.startInstructionDefn();

    dspram.multAxB( imageAddressA + index + 1 + offset, oneB );

  dspram.endInstructionDefn();
  // M0 = Lower, M1 = Mid, M2 = Upper

  dspram.startInstructionDefn();

    dspram.acc0LoadShifter();
    dspram.setShifter( LSL16 );

  dspram.endInstructionDefn();
  // M0 = X, M1 = Lower, M2 = Mid, Acc0 = Upper

  dspram.startInstructionDefn();

    dspram.acc1LoadShifter();
    dspram.setShifter( LSL32 );

  dspram.endInstructionDefn();
  // M0 = X, M1 = X, M2 = Lower, Acc1 = Mid, Acc0 = Upper

  dspram.startInstructionDefn();

    dspram.acc2LoadShifter();
```

```
    dspram.setShifter( noShift );
    dspram.injectOntoBroadcastBus ( numberOfPEs - 1 );
    dspram.regLoadBroadcastBus ();

  dspram.endInstructionDefn();
  // M0 = X, M1 = X, M2 = X, Acc2 = Lower, Acc1 = Mid, Acc0 = Upper

  // Shift PE 0's pixels to the last PE.
  dspram.startInstructionDefn();

    dspram.injectIntoRight( Int48(0) );
    dspram.leftBusFromAcc();
    dspram.accLoadFromRight();
    busValue = dspram.extractFromLeft();

  dspram.endInstructionDefn();

  dspram.startInstructionDefn();

    dspram.pushEnableBeqR( PENumberB );

  dspram.endInstructionDefn();
  // IF PENumberB == numberOfPEs - 1

    dspram.startInstructionDefn();

      dspram.injectIntoRight( busValue );
      dspram.leftBusFromAcc();
      dspram.accLoadFromRight();

    dspram.endInstructionDefn();
    // Acc = PE 0 pixels

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 2, ACC_0 );

    dspram.endInstructionDefn();

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 5, ACC_1 );

    dspram.endInstructionDefn();

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 8, ACC_2 );
      dspram.popEnable();

    dspram.endInstructionDefn();
  // ENDIF

} // if

// Check to see if PE 0 requires pixels to be shifted in.
if ( index % offset )
{

  // Load accumulator with last PE's neighbours
  dspram.startInstructionDefn();

    dspram.multAxB( imageAddressA + index - 1 - offset, oneB );

  dspram.endInstructionDefn();
  // M0 = Upper

  dspram.startInstructionDefn();

    dspram.multAxB( imageAddressA + index - 1, oneB );

  dspram.endInstructionDefn();
  // M0 = Mid, M1 = Upper

  dspram.startInstructionDefn();

    dspram.multAxB( imageAddressA + index - 1 + offset, oneB );

  dspram.endInstructionDefn();
  // M0 = Lower, M1 = Mid, M2 = Upper

  dspram.startInstructionDefn();

    dspram.acc0LoadShifter();
    dspram.setShifter( LSL16 );

  dspram.endInstructionDefn();
  // M0 = X, M1 = Lower, M2 = Mid, Acc0 = Upper

  dspram.startInstructionDefn();

    dspram.acc1LoadShifter();
```

```
    dspram.setShifter( LSL32 );

  dspram.endInstructionDefn();
  // M0 = X, M1 = X, M2 = Lower, Acc1 = Mid, Acc0 = Upper

  dspram.startInstructionDefn();

    dspram.acc2LoadShifter();
    dspram.setShifter( noShift );
    dspram.injectOntoBroadcastBus( 0 );
    dspram.regLoadBroadcastBus();

  dspram.endInstructionDefn();
  // M0 = X, M1 = X, M2 = X, Acc2 = Lower, Acc1 = Mid, Acc0 = Upper

  // Shift PE(N-1)'s pixels to the first PE.
  dspram.startInstructionDefn();

    dspram.injectIntoLeft( Int48(0) );
    dspram.rightBusFromAcc();
    dspram.accLoadFromLeft();
    busValue = dspram.extractFromRight();

  dspram.endInstructionDefn();

  dspram.startInstructionDefn();

    dspram.pushEnableBeqR( PENumberB );

  dspram.endInstructionDefn();
  // IF PENumberB == numberOfPEs - 1

    dspram.startInstructionDefn();

      dspram.injectIntoLeft( busValue );
      dspram.rightBusFromAcc();
      dspram.accLoadFromLeft();

    dspram.endInstructionDefn();
    // Acc = PE(N-1) pixels

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 0, ACC_0 );

    dspram.endInstructionDefn();

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 3, ACC_1 );

    dspram.endInstructionDefn();

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_A, pixelNeighbourhoodA + 6, ACC_2 );
      dspram.popEnable();

    dspram.endInstructionDefn();
  // ENDIF

 } // if

#endif

} // collectNeighbours

void image::load( char * fileName )
{

#ifdef CPP_WORKING

  int   i;
  int   pe;
  int   numberOfPixels;
  char  buffer;
  short shortData;
  Int16 data;

  std::ifstream file( fileName, std::ios::in | std::ios::binary );

  parsePGMHeader( file );
  parsePGMSize( file );
  parsePGMMaxValue( file );

  // Allocate memory for the image in the memory banks

  numberOfPEs = dspram.getNumberOfPEs();
  numberOfPixels = width * height;

  if ( numberOfPixels % (numberOfPEs * 2) )
  {
```

```
    std::cerr << "#_of_PEs_does_not_evenly_divide_into_number_of_pixels" << std::endl;
    abort();

  } // if
  else
  {

    // Calculate the amount of memory required to store the image.
    // Multiply by 2 since there is 2 pixels in a 16-Bit word
    columnImageSize = numberOfPixels / (numberOfPEs * 2);

    // Allocate memory for the original image in BANK_A
    imageAddressA = dspram.intVectorAlloc( BANK_A, columnImageSize );

    // Twice as big for BANK B since there will be 1 pixel per PE
    // in the processed image
    imageAddressB = dspram.intVectorAlloc( BANK_B, columnImageSize * 2 );

    for ( i = 0; i < columnImageSize; i++ )
    {

      for ( pe = 0; pe < numberOfPEs; pe++ )
      {

        file >> buffer;
        shortData = buffer & 0x00FF;
        file >> buffer;

        shortData |= (((short)buffer << 8) & 0xFF00);
        data = Int16( shortData );

        dspram.startInstructionDefn();

          dspram.write( data, pe, BANK_A, imageAddressA + i );

        dspram.endInstructionDefn();

      } // for

    } // for

  } // else

#else

  FILE * fp;

  if ( (fp=fopen( fileName, "rb" )) == NULL )
  {

    std::cerr << "Could_not_open_PGM_file_" << fileName << std::endl;
    abort();

  } // if

  parsePGMHeader( fp );
  parsePGMSize( fp );
  parsePGMMaxValue( fp );

  std::cout << "Image_size_=_" << width << "x" << height << std::endl;

  numberOfPEs = dspram.getNumberOfPEs();
  int numberOfPixels = width * height;

  // Calculate the amount of memory required to store the image.
  // Multiply by 2 since there is 2 pixels in a 16-Bit word
  columnImageSize = numberOfPixels / (numberOfPEs * 2);

  // Allocate memory for the original image in BANK_A
  imageAddressA = dspram.intVectorAlloc( BANK_A, columnImageSize );

  // Twice as big for BANK B since there will be 1 pixel per PE
  // in the processed image
  imageAddressB = dspram.intVectorAlloc( BANK_B, columnImageSize * 2 );

  unsigned short buffer;
  unsigned short endianBuffer;
  int pe;
  int row;
  int i;

  for ( i=0, pe=0, row=0; i<numberOfPixels; i+=2 )
  {

    fread( &buffer, sizeof(buffer), 1, fp );
    endianBuffer  = (buffer & 0x00FF) << 8;
    endianBuffer |= (buffer >> 8) & 0x00FF;

    dspram.write( Int16(endianBuffer), pe, BANK_A, imageAddressA + row );

    pe++;
```

```
    if (pe == numberOfPEs)
    {
      pe = 0;
      row++;
//      std::cout << "Row = " << row << std::endl;
    } // if
//    std::cout << "PE = " << pe << std::endl;
  } // for
  fclose(fp);
#endif
} // load
void image::parsePGMHeader( FILE * fp )
{
  char buffer[128];
  while (true)
  {
    if ( fgets( buffer, 128, fp ) == NULL )
    {
      std::cerr << "Not_a_valid_PGM_file,_no_header" << std::endl;
      abort();
    } // if
    if ( buffer[0] == '#' )
    {
      continue;
    } // if
    else if ( buffer[0] == 'P' )
    {
      if ( buffer[1] == '5' )
      {
        break;
      } // if
      else
      {
        std::cerr << "Invalid_PGM_valid,_wrong_header." << std::endl;
        abort();
      } // else
    } // else if
    else
    {
      std::cerr << "Not_a_valid_PGM_file,_missing_header" << std::endl;
      abort();
    } // else
  } // while
} // parsePGMHeader (C-style)
void image::parsePGMSize( FILE * fp )
{
  char buffer[128];
  while (true)
  {
    if ( fgets( buffer, 128, fp ) == NULL )
    {
      std::cerr << "Not_a_valid_PGM_file,_no_size" << std::endl;
      abort();
    } // if
    if ( buffer[0] == '#' )
    {
```

```
      continue;

    } // if
    else if ( buffer[0] >= '0' || buffer[0] <= '9' )
    {

      int num = sscanf( buffer, "%d_%d\n", &width, &height );

      if ( num <= 1 )
      {

        std::cerr << "Invalid_PGM_file,_corrupt_size" << std::endl;

      } // if
      else
      {

        break;

      } // else

    } // else if
    else
    {

      std::cerr << "Not_a_valid_PGM_file,_missing_size" << std::endl;
      abort();

    } // else

  } // while

} // parsePGMSize (C-style)

void image::parsePGMMaxValue( FILE * fp )
{

  char buffer[128];
  int maxValue;

  while (true)
  {

    if ( fgets( buffer, 128, fp ) == NULL )
    {

      std::cerr << "Not_a_valid_PGM_file,_no_maximum_value" << std::endl;
      abort();

    } // if

    if ( buffer[0] == '#' )
    {

      continue;

    } // if
    else if ( buffer[0] >= '0' || buffer[0] <= '9' )
    {

      int num = sscanf( buffer, "%d\n", &maxValue );

      if ( num <= 0 )
      {

        std::cerr << "Invalid_PGM_file,_corrupt_maximum_value" << std::endl;

      } // if
      else
      {

        break;

      } // else

    } // else if
    else
    {

      std::cerr << "Not_a_valid_PGM_file,_missing_maximum_value" << std::endl;
      abort();

    } // else

  } // while

} // parsePGMMaxValue (C-style)

void image::parsePGMHeader( std::ifstream & file )
{

  char buffer;
```

```
  bool eatline;

  eatline = false;

  while (true)
  {
    file >> buffer;

    if ( buffer == '#' || eatline )
    {

      if ( file.peek() == '\n' )
      {

        eatline = false;

      } // if
      else
      {

        eatline = true;

      } // else

    } // if
    else if ( buffer == 'P' )
    {

      file >> buffer;

      if ( buffer == '5' )
      {

        break;

      } // if
      else
      {

        std::cerr << "Not_a_valid_PGM_file_-_no_header" << std::endl;
        abort();

      } // else

    } // else if
    else
    {

      std::cerr << "Not_a_valid_PGM_file_-_no_header" << std::endl;
      abort();

    } // else

  } // while

} // parsePGMHeader

void image::parsePGMSize( std::ifstream & file )
{

  char buffer;
  bool eatline;

  eatline = false;

  while (true)
  {

    file >> buffer;

    if ( buffer == '#' || eatline )
    {

      if ( file.peek() == '\n' )
      {

        eatline = false;

      } // if
      else
      {

        eatline = true;

      } // else

    } // if
    else if ( buffer >= '0' || buffer <= '9' )
    {

      file.putback( buffer );
```

```
      file >> width >> height;

      std::cerr << width << "x" << height << std::endl;

      break;

    } // else if
    else
    {

      std::cerr << "Not_a_valid_PGM_file_-_no_header" << std::endl;
      abort();

    } // else

  } // while

} // parsePGMSize

void image::parsePGMMaxValue( std::ifstream & file )
{

  char buffer;
  bool eatline;
  int  maxValue;

  eatline = false;

  while (true)
  {

    file >> buffer;

    if ( buffer == '#' || eatline )
    {

      if ( file.peek() == '\n' )
      {

        eatline = false;

      } // if
      else
      {

        eatline = true;

      } // else

    } // if
    else if ( buffer >= '0' || buffer <= '9' )
    {

      file.putback( buffer );

      file >> maxValue;

      std::cerr << maxValue << std::endl;

      break;

    } // else if
    else
    {

      std::cerr << "Not_a_valid_PGM_file_-_no_header" << std::endl;
      abort();

    } // else

  } // while

} // parsePGMMaxValue

void image::compareMax( int addressA, int byte )
{

#ifdef ENHANCED_ROUTING

  dspram.startInstructionDefn();

    dspram.enableByteSelect( byte );
    dspram.pushEnableAgtB( addressA, comparisonPixelB );
    dspram.saveInputA( BANK_A, addressA );

  dspram.endInstructionDefn( DEBUG );
  // IF pixel > max THEN

    dspram.startInstructionDefn();

      dspram.memLoadInputA( BANK_B, comparisonPixelB );
```

```
      dspram.popEnable();

    dspram.endInstructionDefn( DEBUG );

  // END IF

#else

  dspram.startInstructionDefn();

    dspram.enableByteSelect( byte );
    dspram.pushEnableAgtB( addressA, comparisonPixelB );

  dspram.endInstructionDefn();
  // IF pixel > max THEN

    dspram.startInstructionDefn();

      dspram.multAxB( addressA, oneB );

    dspram.endInstructionDefn();
    // M0 = max

    dspram.startInstructionDefn();
    dspram.endInstructionDefn();
    // M0 = X, M1 = max

    dspram.startInstructionDefn();
    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = max

    dspram.startInstructionDefn();

      dspram.accLoadShifter();

    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = X, Acc = max

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_B, comparisonPixelB, ACC_0 );
      dspram.popEnable();

    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = X, Mem = max

  // END IF

#endif

} // compareMax

void image::compareMin( int addressA, int byte )
{

#ifdef ENHANCED_ROUTING

  dspram.startInstructionDefn();

    dspram.enableByteSelect( byte );
    dspram.pushEnableAltB( addressA, comparisonPixelB );
    dspram.saveInputA( BANK_A, addressA );

  dspram.endInstructionDefn( DEBUG );
  // IF pixel > max THEN

    dspram.startInstructionDefn();

      dspram.memLoadInputA( BANK_B, comparisonPixelB );
      dspram.popEnable();

    dspram.endInstructionDefn( DEBUG );

  // END IF

#else

  dspram.startInstructionDefn();

    dspram.enableByteSelect( byte );
    dspram.pushEnableAltB( addressA, comparisonPixelB );

  dspram.endInstructionDefn();
  // IF pixel < min THEN

    dspram.startInstructionDefn();

      dspram.multAxB( addressA, oneB );

    dspram.endInstructionDefn();
    // M0 = min
```

```
    dspram.startInstructionDefn();
    dspram.endInstructionDefn();
    // M0 = X, M1 = min

    dspram.startInstructionDefn();
    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = min

    dspram.startInstructionDefn();

      dspram.accLoadShifter();

    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = X, Acc = min

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_B, comparisonPixelB , ACC_0 );
      dspram.popEnable();

    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = X, Mem = min

  // END IF

#endif

} // compareMin

void image::dumpNeighbourhood( int pe )
{

  std::cout << std::hex << std::endl;
  std::cout << dspram.read( pe, BANK_A, pixelNeighbourhoodA + 0 ) << "_"
            << dspram.read( pe, BANK_A, pixelNeighbourhoodA + 1 ) << "_"
            << dspram.read( pe, BANK_A, pixelNeighbourhoodA + 2 ) << std::endl
            << dspram.read( pe, BANK_A, pixelNeighbourhoodA + 3 ) << "_"
            << dspram.read( pe, BANK_A, pixelNeighbourhoodA + 4 ) << "_"
            << dspram.read( pe, BANK_A, pixelNeighbourhoodA + 5 ) << std::endl
            << dspram.read( pe, BANK_A, pixelNeighbourhoodA + 6 ) << "_"
            << dspram.read( pe, BANK_A, pixelNeighbourhoodA + 7 ) << "_"
            << dspram.read( pe, BANK_A, pixelNeighbourhoodA + 8 ) << std::endl;

  std::cout << std::dec << std::endl;

} // dumpNeighbourhood

void image::sortNeighbourhood( void )
{

  int i;
  int j;

#if DEBUG
  int pe;

  pe = 1;

// Dump sorting neighbourhood

  std::cout << std::hex << std::endl;
  std::cout << (short)dspram.read( pe, BANK_B, sortingArrayB + 0 ) << "_"
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 1 ) << "_"
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 2 ) << std::endl
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 3 ) << "_"
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 4 ) << "_"
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 5 ) << std::endl
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 6 ) << "_"
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 7 ) << "_"
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 8 ) << std::endl;

  std::cout << std::dec << std::endl;
#endif

  for ( i=0; i<5; i++ )
  {

    for ( j=0; j<8; j++ )
    {

#ifdef ENHANCED_ROUTING

      // Swap pixels
      dspram.startInstructionDefn();

        dspram.disableByteSelect();
        dspram.saveInputA( BANK_B, sortingArrayB + j );
        dspram.regLoadDataB( sortingArrayB + j );

      dspram.endInstructionDefn(DEBUG);
      // MA = B(j), R = B(j)
```

```
        dspram.startInstructionDefn();

          dspram.pushEnableBltR( sortingArrayB + j + 1 );
          dspram.memLoadInputA( BANK_A, temporaryStorageA );
          dspram.saveInputA( BANK_B, sortingArrayB + j + 1 );

        dspram.endInstructionDefn(DEBUG);
        // IF B(j+1) < B(j), temp = B(j), MA = B(j+1)

          // Swap the pixels
          dspram.startInstructionDefn();

            dspram.saveInputA( BANK_A, temporaryStorageA );
            dspram.memLoadInputA( BANK_B, sortingArrayB + j );

          dspram.endInstructionDefn(DEBUG);
          // B(j) = B(j+1), MA = temp

          dspram.startInstructionDefn();

            dspram.memLoadInputA( BANK_B, sortingArrayB + j + 1 );
            dspram.popEnable();

          dspram.endInstructionDefn(DEBUG);
          // B(j+1) = temp

        // ENDIF

#else

        // Swap pixels
        dspram.startInstructionDefn();

          dspram.regLoadDataB( sortingArrayB + j );

        dspram.endInstructionDefn(DEBUG);
        // R = B0

        dspram.startInstructionDefn();

          dspram.pushEnableBltR( sortingArrayB + j + 1 );

        dspram.endInstructionDefn(DEBUG);
        // IF B(j+1) < B(j)

          // Swap the pixels
          dspram.startInstructionDefn();

            dspram.disableByteSelect();
            dspram.disableAdder();
            dspram.multAxB( oneA, sortingArrayB + j );

          dspram.endInstructionDefn(DEBUG);
          // M0 = B(j)

          dspram.startInstructionDefn();

            dspram.multAxB( oneA, sortingArrayB + j + 1 );

          dspram.endInstructionDefn(DEBUG);
          // M0 = B(j+1), M1 = B(j)

          dspram.startInstructionDefn();
          dspram.endInstructionDefn(DEBUG);
          // M0 = X, M1 = B(j+1), M2 = B(j)

          dspram.startInstructionDefn();

            dspram.accLoadShifter();

          dspram.endInstructionDefn(DEBUG);
          // M0 = X, M1 = X, M2 = B(j+1), Acc = B(j)

          dspram.startInstructionDefn();

            dspram.accLoadShifter();
            dspram.memLoadAccWord( BANK_B, sortingArrayB + j + 1, ACC_0 );

          dspram.endInstructionDefn(DEBUG);
          // M0 = X, M1 = X, M2 = X, Acc = B(j+1), B(j+1) = B(j)

          dspram.startInstructionDefn();

            dspram.memLoadAccWord( BANK_B, sortingArrayB + j, ACC_0 );
            dspram.popEnable();

          dspram.endInstructionDefn(DEBUG);
          // M0 = X, M1 = X, M2 = X, B(j) = B(j+1), B(j+1) = B(j)

        // ENDIF

#endif
```

```
    } // for
  } // for
#if DEBUG
  std::cout << std::hex << std::endl;
  std::cout << (short)dspram.read( pe, BANK_B, sortingArrayB + 0 ) << "_"
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 1 ) << "_"
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 2 ) << std::endl
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 3 ) << "_"
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 4 ) << "_"
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 5 ) << std::endl
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 6 ) << "_"
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 7 ) << "_"
            << (short)dspram.read( pe, BANK_B, sortingArrayB + 8 ) << std::endl;

  std::cout << std::dec << std::endl;

  abort();
#endif
} // sortNeighbourhood
```

## Listing D.4: Threshold

```
// $Id: threshold.cc,v 1.2 2002/07/23 21:28:45 dillen Exp $
//
// $Log: threshold.cc,v $
// Revision 1.2  2002/07/23 21:28:45  dillen
// Main program for image thresholding
//
// Revision 1.1  2002/07/22 21:21:59  dillen
// Initial revision
//

#include <iostream>
#include <dspramClass.h>
#include "../image/image.h"

#define DEBUG 0

void image::threshold( Int16 minimum, Int16 maximum, Int16 thresholdValue )
{
  int i;

  // Step 1: Store the minimum value into ACC0
  // Step 2: Store the maximum value into ACC1
  // Step 3: Store the threshold value into Broadcast Bus Register
  // Step 4: Compare all pixel values to threshold value
  // Step 5: If pixel is GE threshold store max
  // Step 6: If pixel is LT threshold store min

  // Broadcast the minimum value
  dspram.startInstructionDefn();

    dspram.injectOntoBroadcastBus ( minimum );
    dspram.regLoadBroadcastBus();
    dspram.disableAdder();
    dspram.setShifter( noShift );

  dspram.endInstructionDefn();

  // Broadcast the maximum value, multiply the minimum value
  dspram.startInstructionDefn();

    dspram.multBxR( oneB );
    dspram.injectOntoBroadcastBus ( maximum );
    dspram.regLoadBroadcastBus();

  dspram.endInstructionDefn();
  // M0 = Minimum

  dspram.startInstructionDefn();

    dspram.multBxR( oneB );
    dspram.injectOntoBroadcastBus ( thresholdValue );
    dspram.regLoadBroadcastBus();

  dspram.endInstructionDefn();
  // M0 = Maximum, M1 = Minimum

  dspram.startInstructionDefn();
  dspram.endInstructionDefn();
  // M0 = X, M1 = Maximum, Mout = Minimum

  dspram.startInstructionDefn();
```

```
    dspram.acc0LoadShifter();
    dspram.setShifter( LSL16 );

  dspram.endInstructionDefn();
  // M0 = X, M1 = X, Mout = Maximum, Acc0 = Threshold

  dspram.startInstructionDefn();

    dspram.acc1LoadShifter();
    dspram.setShifter( noShift );

  dspram.endInstructionDefn();
  // M0 = X, M1 = X, Mout = X, Acc1 = Maximum, Acc0 = Minimum

  // For each pixel in memory, compare the threshold
  for ( i = 0; i < columnImageSize; i++ )
  {

    // IF pixel >= threshold THEN
    dspram.startInstructionDefn();

      dspram.enableByteSelect( 0 );        // Compare lower byte first
      dspram.pushEnableAgeR( imageAddressA + i );
      dspram.multAxR( imageAddressA + i );

    dspram.endInstructionDefn(DEBUG);

      dspram.startInstructionDefn();

        dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i, ACC_1 );
        dspram.toggleEnable();

      dspram.endInstructionDefn(DEBUG);

    // ELSE

      dspram.startInstructionDefn();

        dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i, ACC_0 );
        dspram.popEnable();

      dspram.endInstructionDefn(DEBUG);

    // END IF

    // IF pixel >= threshold THEN
    dspram.startInstructionDefn();

      dspram.enableByteSelect(1);         // Compare upper byte
      dspram.pushEnableAgeR( imageAddressA + i );
      dspram.multAxR( imageAddressA + i );

    dspram.endInstructionDefn(DEBUG);

      dspram.startInstructionDefn();

        dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i + 1, ACC_1 );
        dspram.toggleEnable();

      dspram.endInstructionDefn(DEBUG);

    // ELSE

      dspram.startInstructionDefn();

        dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i + 1, ACC_0 );
        dspram.popEnable();

      dspram.endInstructionDefn(DEBUG);

    // ENDIF

  } // for

} // threshold
```

## Listing D.5: Sobel Dx and Dy

```
// $Id: filter3x3.cc,v 1.2 2002/07/29 20:33:09 dillen Exp dillen $
//
// $Log: filter3x3.cc,v $
// Revision 1.2  2002/07/29 20:33:09  dillen
// 3x3 Filter
//
// Revision 1.1  2002/07/23 21:37:38  dillen
// Initial revision
//
//
```

```
#include <iostream>
#include <dspramClass.h>
#include "../image/image.h"

void image::filter3x3( Int16 coefficients[9] )
{

  Int i;
  Int offset;

#if DEBUG
  Int pe = 1;
#endif

  // Step 1: Shift pixels to left and right neighbours
  // Step 2: MAC the pixel neighbourhood with the filter coefficients for lower pixel
  // Step 3: Store the lower pixel in BANK B
  // Step 4: MAC the pixel neighbourhood with the filter coefficients for upper pixel
  // Step 5: Store the upper pixel in BANK B

  // Calculate the offset
  offset = width / (numberOfPEs * 2);

#if DEBUG

  std::cout << "Offset = " << offset << std::endl;

  std::cout << "Actual neighbours (before collection):" << std::endl;

  // Dump image pixels, and neighbours
  std::cout << std::hex << (short)dspram.read( pe - 1, BANK_A, imageAddressA              ) << " "
            << std::hex << (short)dspram.read( pe,     BANK_A, imageAddressA              ) << " "
            << std::hex << (short)dspram.read( pe + 1, BANK_A, imageAddressA              ) << std::endl
            << std::hex << (short)dspram.read( pe - 1, BANK_A, imageAddressA + 1 * offset ) << " "
            << std::hex << (short)dspram.read( pe,     BANK_A, imageAddressA + 1 * offset ) << " "
            << std::hex << (short)dspram.read( pe + 1, BANK_A, imageAddressA + 1 * offset ) << std::endl
            << std::hex << (short)dspram.read( pe - 1, BANK_A, imageAddressA + 2 * offset ) << " "
            << std::hex << (short)dspram.read( pe,     BANK_A, imageAddressA + 2 * offset ) << " "
            << std::hex << (short)dspram.read( pe + 1, BANK_A, imageAddressA + 2 * offset ) << std::endl;

#endif

  for ( i = offset; i < (columnImageSize - offset); i++ )
  {

    collectNeighbours( i );

#if DEBUG

    std::cout << std::endl << "Collected Neighbours:" << std::endl;

    dumpNeighbourhood(pe);

#endif

    // Process the lower pixel
    dspram.startInstructionDefn();

      dspram.injectOntoBroadcastBus( coefficients[0] );
      dspram.regLoadBroadcastBus();
      dspram.accClear();
      dspram.disableAdder();

    dspram.endInstructionDefn(DEBUG);

    dspram.startInstructionDefn();

      dspram.injectOntoBroadcastBus( coefficients[1] );
      dspram.regLoadBroadcastBus();
      dspram.enableByteSelect( 1 );     // Upper Left pixel is in upper byte
      dspram.multAxR( pixelNeighbourhoodA + 0 );

    dspram.endInstructionDefn(DEBUG);
    // M0 = UL

    dspram.startInstructionDefn();

      dspram.injectOntoBroadcastBus( coefficients[2] );
      dspram.regLoadBroadcastBus();
      dspram.enableByteSelect( 0 );     // Upper Mid pixel is in lower byte
      dspram.multAxR( pixelNeighbourhoodA + 1 );

    dspram.endInstructionDefn(DEBUG);
    // M0 = UM, M1 = UL

    dspram.startInstructionDefn();

      dspram.injectOntoBroadcastBus( coefficients[3] );
      dspram.regLoadBroadcastBus();
      dspram.enableByteSelect( 1 );     // Upper Right pixel is in upper byte
      dspram.multAxR( pixelNeighbourhoodA + 1 );
```

```
dspram.endInstructionDefn(DEBUG);
// M0 = UR, M1 = UM, M2 = UL

dspram.startInstructionDefn();

  dspram.injectOntoBroadcastBus( coefficients[4] );
  dspram.regLoadBroadcastBus();
  dspram.enableByteSelect( 1 );     // Mid left pixel is in upper byte
  dspram.multAxR( pixelNeighbourhoodA + 3 );
  dspram.accLoadShifter();
  dspram.enableAdder();

dspram.endInstructionDefn(DEBUG);
// M0 = ML, M1 = UR, M2 = UM, ACC = UL

dspram.startInstructionDefn();

  dspram.injectOntoBroadcastBus( coefficients[5] );
  dspram.regLoadBroadcastBus();
  dspram.enableByteSelect( 0 );     // Mid mid pixel is in lower byte
  dspram.multAxR( pixelNeighbourhoodA + 4 );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = MM, M1 = ML, M2 = UR, ACC = UL+UM

dspram.startInstructionDefn();

  dspram.injectOntoBroadcastBus( coefficients[6] );
  dspram.regLoadBroadcastBus();
  dspram.enableByteSelect( 1 );     // Mid right pixel is in upper byte
  dspram.multAxR( pixelNeighbourhoodA + 4 );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = MR, M1 = MM, M2 = ML, ACC = UL+UM+UR

dspram.startInstructionDefn();

  dspram.injectOntoBroadcastBus( coefficients[7] );
  dspram.regLoadBroadcastBus();
  dspram.enableByteSelect( 1 );     // Lower left pixel is in upper byte
  dspram.multAxR( pixelNeighbourhoodA + 6 );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = LL, M1 = MR, M2 = MM, ACC = UL+UM+UR+ML

dspram.startInstructionDefn();

  dspram.injectOntoBroadcastBus( coefficients[8] );
  dspram.regLoadBroadcastBus();
  dspram.enableByteSelect( 0 );     // Lower mid pixel is in lower byte
  dspram.multAxR( pixelNeighbourhoodA + 7 );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = LM, M1 = LL, M2 = MR, ACC = UL+UM+UR+ML+MM

dspram.startInstructionDefn();

  dspram.injectOntoBroadcastBus( coefficients[0] );
  dspram.regLoadBroadcastBus();
  dspram.enableByteSelect( 1 );     // Lower right pixel is in upper byte
  dspram.multAxR( pixelNeighbourhoodA + 7 );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = LR, M1 = LM, M2 = LL, ACC = UL+UM+UR+ML+MM+MR

dspram.startInstructionDefn();

  dspram.injectOntoBroadcastBus( coefficients[1] );
  dspram.regLoadBroadcastBus();
  dspram.enableByteSelect( 0 );     // Upper left pixel is in lower byte
  dspram.multAxR( pixelNeighbourhoodA + 1 );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = UL, M1 = LR, M2 = LM, ACC = UL+UM+UR+ML+MM+MR+LL

dspram.startInstructionDefn();

  dspram.injectOntoBroadcastBus( coefficients[2] );
  dspram.regLoadBroadcastBus();
  dspram.enableByteSelect( 1 );     // Upper mid pixel is in upper byte
  dspram.multAxR( pixelNeighbourhoodA + 1 );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = UM, M1 = UL, M2 = LR, ACC = UL+UM+UR+ML+MM+MR+LL+LM
```

```
    dspram.startInstructionDefn();

      dspram.injectOntoBroadcastBus( coefficients[3] );
      dspram.regLoadBroadcastBus();
      dspram.enableByteSelect( 0 );     // Upper right pixel is in lower byte
      dspram.multAxR( pixelNeighbourhoodA + 2 );
      dspram.accLoadShifter();
      dspram.disableAdder();

    dspram.endInstructionDefn(DEBUG);
    // M0 = UR, M1 = UM, M2 = UL, ACC = UL+UM+UR+ML+MM+MR+LL+LM+LR

#if DEBUG
std::cout << "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@" << std::endl;
#endif

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i + 0, ACC.0 );
      dspram.injectOntoBroadcastBus( coefficients[4] );
      dspram.regLoadBroadcastBus();
      dspram.enableByteSelect( 0 );     // Mid left pixel is in lower byte
      dspram.multAxR( pixelNeighbourhoodA + 4 );
      dspram.accLoadShifter();
      dspram.enableAdder();

    dspram.endInstructionDefn(DEBUG);
    // M0 = ML, M1 = UR, M2 = UM, ACC = UL

#if DEBUG
std::cout << "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@" << std::endl;
#endif

    dspram.startInstructionDefn();

      dspram.injectOntoBroadcastBus( coefficients[5] );
      dspram.regLoadBroadcastBus();
      dspram.enableByteSelect( 1 );     // Mid mid pixel is in upper byte
      dspram.multAxR( pixelNeighbourhoodA + 4 );
      dspram.accLoadShifter();

    dspram.endInstructionDefn(DEBUG);
    // M0 = MM, M1 = ML, M2 = UR, ACC = UL+UM

    dspram.startInstructionDefn();

      dspram.injectOntoBroadcastBus( coefficients[6] );
      dspram.regLoadBroadcastBus();
      dspram.enableByteSelect( 0 );     // Mid right pixel is in lower byte
      dspram.multAxR( pixelNeighbourhoodA + 5 );
      dspram.accLoadShifter();

    dspram.endInstructionDefn(DEBUG);
    // M0 = MR, M1 = MM, M2 = ML, ACC = UL+UM+UR

    dspram.startInstructionDefn();

      dspram.injectOntoBroadcastBus( coefficients[7] );
      dspram.regLoadBroadcastBus();
      dspram.enableByteSelect( 0 );     // Lower left pixel is in lower byte
      dspram.multAxR( pixelNeighbourhoodA + 7 );
      dspram.accLoadShifter();

    dspram.endInstructionDefn(DEBUG);
    // M0 = LL, M1 = MR, M2 = MM, ACC = UL+UM+UR+ML

    dspram.startInstructionDefn();

      dspram.injectOntoBroadcastBus( coefficients[8] );
      dspram.regLoadBroadcastBus();
      dspram.enableByteSelect( 1 );     // Lower mid pixel is in upper byte
      dspram.multAxR( pixelNeighbourhoodA + 7 );
      dspram.accLoadShifter();

    dspram.endInstructionDefn(DEBUG);
    // M0 = LM, M1 = LL, M2 = MR, ACC = UL+UM+UR+ML+MM

    dspram.startInstructionDefn();

      dspram.enableByteSelect( 0 );     // Lower right pixel is in lower byte
      dspram.multAxR( pixelNeighbourhoodA + 8 );
      dspram.accLoadShifter();

    dspram.endInstructionDefn(DEBUG);
    // M0 = LR, M1 = LM, M2 = LL, ACC = UL+UM+UR+ML+MM+MR

    dspram.startInstructionDefn();

      dspram.accLoadShifter();

    dspram.endInstructionDefn(DEBUG);
    // M0 = X, M1 = LR, M2 = LM, ACC = UL+UM+UR+ML+MM+MR+LL
```

```
    dspram.startInstructionDefn();

      dspram.accLoadShifter();

    dspram.endInstructionDefn(DEBUG);
    // M0 = X, M1 = X, M2 = LR, ACC = UL+UM+UR+ML+MM+MR+LL+LM

    dspram.startInstructionDefn();

      dspram.accLoadShifter();

    dspram.endInstructionDefn(DEBUG);
    // M0 = X, M1 = X, M2 = X, ACC = UL+UM+UR+ML+MM+MR+LL+LM+LR

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i + 1, ACC.0 );
      dspram.disableByteSelect();

    dspram.endInstructionDefn(DEBUG);
    // Store the upper pixel value

    // Output the processed pixel

#if DEBUG

    std::cout << std::endl << "Processed pixel value = "
              << dspram.read( pe, BANK_B, imageAddressB + i * 2 ) << "_"
              << dspram.read( pe, BANK_B, imageAddressB + i * 2 + 1 ) << std::endl;

    abort();

#endif

//    abort();

  } // for

} // filter3x3
```

## Listing D.6: Box Filter

```
// $Id: boxFilter.cc,v 1.1 2002/07/29 20:32:30 dillen Exp $
//
// $Log: boxFilter.cc,v $
// Revision 1.1  2002/07/29 20:32:30  dillen
// Initial revision
//

#include <iostream>
#include <dspramClass.h>
#include "../image/image.h"

#define ONE_NINTH_16 (0x1C70)  // 1/9 shifted 16 bits
#define DEBUG 0

void image::box( void )
{

  int i;
  int offset;

  // Step 1: Shift pixels to left and right neighbours
  // Step 2: MAC the pixel neighbourhood with the filter coefficients for lower pixel
  // Step 3: Store the lower pixel in BANK B
  // Step 4: MAC the pixel neighbourhood with the filter coefficients for upper pixel
  // Step 5: Store the upper pixel in BANK B

  // Calculate the offset
  offset = width / (numberOfPEs * 2);

  for ( i = offset; i < (columnImageSize - offset); i++ )
  {

    collectNeighbours( i );

    // Process the lower pixel
    dspram.startInstructionDefn();

      dspram.enableByteSelect( 1 );     // Upper Left pixel is in upper byte
      dspram.multAxB( pixelNeighbourhoodA + 0, oneB );
      dspram.accClear();
      dspram.disableAdder();
      dspram.setShifter( noShift );

    dspram.endInstructionDefn(DEBUG);
    // M0 = UL

    dspram.startInstructionDefn();
```

```
  dspram.enableByteSelect( 0 );     // Upper Mid pixel is in lower byte
  dspram.multAxB( pixelNeighbourhoodA + 1, oneB );

dspram.endInstructionDefn(DEBUG);
// M0 = UM, M1 = UL

dspram.startInstructionDefn();

  dspram.enableByteSelect( 1 );     // Upper Right pixel is in upper byte
  dspram.multAxB( pixelNeighbourhoodA + 1, oneB );

dspram.endInstructionDefn(DEBUG);
// M0 = UR, M1 = UM, M2 = UL

dspram.startInstructionDefn();

  dspram.enableByteSelect( 1 );     // Mid left pixel is in upper byte
  dspram.multAxB( pixelNeighbourhoodA + 3, oneB );
  dspram.accLoadShifter();
  dspram.enableAdder();

dspram.endInstructionDefn(DEBUG);
// M0 = ML, M1 = UR, M2 = UM, ACC = UL

dspram.startInstructionDefn();

  dspram.enableByteSelect( 0 );     // Mid mid pixel is in lower byte
  dspram.multAxB( pixelNeighbourhoodA + 4, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = MM, M1 = ML, M2 = UR, ACC = UL+UM

dspram.startInstructionDefn();

  dspram.enableByteSelect( 1 );     // Mid right pixel is in upper byte
  dspram.multAxB( pixelNeighbourhoodA + 4, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = MR, M1 = MM, M2 = ML, ACC = UL+UM+UR

dspram.startInstructionDefn();

  dspram.enableByteSelect( 1 );     // Lower left pixel is in upper byte
  dspram.multAxB( pixelNeighbourhoodA + 6, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = LL, M1 = MR, M2 = MM, ACC = UL+UM+UR+ML

dspram.startInstructionDefn();

  dspram.enableByteSelect( 0 );     // Lower mid pixel is in lower byte
  dspram.multAxB( pixelNeighbourhoodA + 7, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = LM, M1 = LL, M2 = MR, ACC = UL+UM+UR+ML+MM

dspram.startInstructionDefn();

  dspram.enableByteSelect( 1 );     // Lower right pixel is in upper byte
  dspram.multAxB( pixelNeighbourhoodA + 7, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = LR, M1 = LM, M2 = LL, ACC = UL+UM+UR+ML+MM+MR

// Start Processing the upper pixel
dspram.startInstructionDefn();

  dspram.enableByteSelect( 0 );     // Upper left pixel is in lower byte
  dspram.multAxB( pixelNeighbourhoodA + 1, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = UL, M1 = LR, M2 = LM, ACC = UL+UM+UR+ML+MM+MR+LL

dspram.startInstructionDefn();

  dspram.enableByteSelect( 1 );     // Upper mid pixel is in upper byte
  dspram.multAxB( pixelNeighbourhoodA + 1, oneB );
  dspram.accLoadShifter();
  dspram.setShifter( LSL16 );

dspram.endInstructionDefn(DEBUG);
// M0 = UM, M1 = UL, M2 = LR, ACC = UL+UM+UR+ML+MM+MR+LL+LM

dspram.startInstructionDefn();
```

```
  dspram.enableByteSelect( 0 );     // Upper right pixel is in lower byte
  dspram.multAxB( pixelNeighbourhoodA + 2, oneB );
  dspram.accLoadShifter();
  dspram.setShifter( noShift );

dspram.endInstructionDefn(DEBUG);
// M0 = UR, M1 = UM, M2 = UL, ACC = UL+UM+UR+ML+MM+MR+LL+LM+LR

// ACC1 = Sum of lower pixel
dspram.startInstructionDefn();

  dspram.enableByteSelect( 0 );     // Mid left pixel is in lower byte
  dspram.multAxB( pixelNeighbourhoodA + 4, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = ML, M1 = UR, M2 = UM, ACC = UL, ACC1 = lowerSum

dspram.startInstructionDefn();

  dspram.enableByteSelect( 1 );     // Mid mid pixel is in upper byte
  dspram.multAxB( pixelNeighbourhoodA + 4, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = MM, M1 = ML, M2 = UR, ACC = UL+UM, ACC1 = lowerSum

dspram.startInstructionDefn();

  dspram.enableByteSelect( 0 );     // Mid right pixel is in lower byte
  dspram.multAxB( pixelNeighbourhoodA + 5, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = MR, M1 = MM, M2 = ML, ACC = UL+UM+UR, ACC1 = lowerSum

dspram.startInstructionDefn();

  dspram.enableByteSelect( 0 );     // Lower left pixel is in lower byte
  dspram.multAxB( pixelNeighbourhoodA + 7, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = LL, M1 = MR, M2 = MM, ACC = UL+UM+UR+ML, ACC1 = lowerSum

dspram.startInstructionDefn();

  dspram.enableByteSelect( 1 );     // Lower mid pixel is in upper byte
  dspram.multAxB( pixelNeighbourhoodA + 7, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = LM, M1 = LL, M2 = MR, ACC = UL+UM+UR+ML+MM, ACC1 = lowerSum

dspram.startInstructionDefn();

  dspram.enableByteSelect( 0 );     // Lower right pixel is in lower byte
  dspram.multAxB( pixelNeighbourhoodA + 8, oneB );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = LR, M1 = LM, M2 = LL, ACC = UL+UM+UR+ML+MM+MR, ACC1 = lowerSum

dspram.startInstructionDefn();

  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = X, M1 = LR, M2 = LM, ACC = UL+UM+UR+ML+MM+MR+LL, ACC1 = lowerSum

dspram.startInstructionDefn();

  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = X, M1 = X, M2 = LR, ACC = UL+UM+UR+ML+MM+MR+LL+LM, ACC1 = lowerSum

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, boxFilterLowerA, ACC_1 );
  dspram.accLoadShifter();

dspram.endInstructionDefn(DEBUG);
// M0 = X, M1 = X, M2 = X, ACC = UL+UM+UR+ML+MM+MR+LL+LM+LR, ACC1 = lowerSum

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_A, boxFilterUpperA, ACC_0 );
  dspram.injectOntoBroadcastBus ( ONE_NINTH_16 );
  dspram.regLoadBroadcastBus();
  dspram.disableByteSelect();
```

```
    dspram.endInstructionDefn(DEBUG);

    dspram.startInstructionDefn();

      dspram.multAxR( boxFilterLowerA );
      dspram.disableAdder();
      dspram.setShifter( ASR16 );

    dspram.endInstructionDefn(DEBUG);
    // M0 = lower

    dspram.startInstructionDefn();

      dspram.multAxR( boxFilterUpperA );

    dspram.endInstructionDefn(DEBUG);
    // M0 = upper, M1 = lower

    dspram.startInstructionDefn();
    dspram.endInstructionDefn(DEBUG);
    // M0 = X, M1 = upper, M2 = lower

    dspram.startInstructionDefn();

      dspram.accLoadShifter();

    dspram.endInstructionDefn(DEBUG);
    // M0 = X, M1 = X, M2 = upper, Acc = lower

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_B, imageAddressB + i * 2 + 0, ACC_0 );
      dspram.accLoadShifter();

    dspram.endInstructionDefn(DEBUG);
    // M0 = X, M1 = X, M2 = X, Acc = upper, Mem = lower

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_B, imageAddressB + i * 2 + 1, ACC_0 );
      dspram.setShifter( noShift );

    dspram.endInstructionDefn(DEBUG);
    // M0 = X, M1 = X, M2 = X, Acc = upper, Mem = upper

#if DEBUG

    abort();

#endif

  } // for

} // boxFilter
```

## Listing D.7: Dilate

```
// $Id: dilate.cc,v 1.1 2002/07/29 20:21:17 dillen Exp dillen $
//
// $Log: dilate.cc,v $
// Revision 1.1  2002/07/29 20:21:17  dillen
// Initial revision
//

#include <iostream>
#include <dspramClass.h>
#include "../image/image.h"

void image::dilate( void )
{

  int i;
  int offset;

  // Step 1: Shift pixels to left and right neighbours
  // Step 2: Find the largest pixel value for lower pixel
  // Step 3: Store the lower pixel value in BANK B
  // Step 4: Find the smallest pixel value for the upper pixel
  // Step 5: Store the upper pixel in BANK B

  // Calculate the offset
  offset = width / (numberOfPEs * 2);

  for ( i = offset; i < (columnImageSize - offset); i++ )
  {

    collectNeighbours( i );

#if DEBUG
```

```
    dumpNeighbourhood( 1 );
#endif

#ifdef ENHANCED_ROUTING

    // Store the first neighbour into the maximum location
    dspram.startInstructionDefn();

      dspram.enableByteSelect( 1 );
      dspram.saveInputA( BANK_A, pixelNeighbourhoodA );

    dspram.endInstructionDefn(DEBUG);

    dspram.startInstructionDefn();

      dspram.memLoadInputA( BANK_B, comparisonPixelB );

    dspram.endInstructionDefn(DEBUG);

    compareMax( pixelNeighbourhoodA + 1, 0 );
    compareMax( pixelNeighbourhoodA + 1, 1 );
    compareMax( pixelNeighbourhoodA + 3, 1 );
    compareMax( pixelNeighbourhoodA + 4, 0 );
    compareMax( pixelNeighbourhoodA + 4, 1 );
    compareMax( pixelNeighbourhoodA + 6, 1 );
    compareMax( pixelNeighbourhoodA + 7, 0 );
    compareMax( pixelNeighbourhoodA + 7, 1 );

    // Save the maximum value to address B
    dspram.startInstructionDefn();

      dspram.disableByteSelect();
      dspram.saveInputA( BANK_B, comparisonPixelB );

    dspram.endInstructionDefn(DEBUG);

    dspram.startInstructionDefn();

      dspram.memLoadInputA( BANK_B, imageAddressB + 2 * i + 0 );

    dspram.endInstructionDefn(DEBUG);

    // Process the upper pixel
    dspram.startInstructionDefn();

      dspram.enableByteSelect( 0 );
      dspram.saveInputA( BANK_A, pixelNeighbourhoodA + 1 );

    dspram.endInstructionDefn(DEBUG);

    dspram.startInstructionDefn();

      dspram.memLoadInputA( BANK_B, comparisonPixelB );

    dspram.endInstructionDefn(DEBUG);

    compareMax( pixelNeighbourhoodA + 1, 1 );
    compareMax( pixelNeighbourhoodA + 2, 0 );
    compareMax( pixelNeighbourhoodA + 4, 0 );
    compareMax( pixelNeighbourhoodA + 4, 1 );
    compareMax( pixelNeighbourhoodA + 5, 0 );
    compareMax( pixelNeighbourhoodA + 7, 0 );
    compareMax( pixelNeighbourhoodA + 7, 1 );
    compareMax( pixelNeighbourhoodA + 8, 0 );

    // Save the maximum value to address B
    dspram.startInstructionDefn();

      dspram.disableByteSelect();
      dspram.saveInputA( BANK_B, comparisonPixelB );

    dspram.endInstructionDefn(DEBUG);

    dspram.startInstructionDefn();

      dspram.memLoadInputA( BANK_B, imageAddressB + 2 * i + 1 );

    dspram.endInstructionDefn(DEBUG);

#else

    // Store the first neighbour into the maximum location
    dspram.startInstructionDefn();

      dspram.disableAdder();
      dspram.setShifter( noShift );
      dspram.enableByteSelect( 1 );  // Upper byte
      dspram.multAxB( pixelNeighbourhoodA, oneB );

    dspram.endInstructionDefn( DEBUG );
    // M0 = p(0)
```

```
// Nops
dspram.startInstructionDefn();
dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = p(0)

dspram.startInstructionDefn();
dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = p(0)

dspram.startInstructionDefn();

  dspram.accLoadShifter();

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Acc = p(0)

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_B, comparisonPixelB , ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Mem = p(0)

compareMax( pixelNeighbourhoodA + 1 , 0 );
compareMax( pixelNeighbourhoodA + 1 , 1 );
compareMax( pixelNeighbourhoodA + 3 , 1 );
compareMax( pixelNeighbourhoodA + 4 , 0 );
compareMax( pixelNeighbourhoodA + 4 , 1 );
compareMax( pixelNeighbourhoodA + 6 , 1 );
compareMax( pixelNeighbourhoodA + 7 , 0 );
compareMax( pixelNeighbourhoodA + 7 , 1 );

// Store the maximum value
dspram.startInstructionDefn();

  dspram.disableByteSelect();
  dspram.multAxB( oneA , comparisonPixelB );

dspram.endInstructionDefn( DEBUG );
// M0 = max

dspram.startInstructionDefn();

  dspram.enableByteSelect( 0 );
  dspram.multAxB( pixelNeighbourhoodA + 1 , oneB );

dspram.endInstructionDefn( DEBUG );
// M0 = pixel0 , M1 = max

dspram.startInstructionDefn();
dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = pixel0 , M2 = max

dspram.startInstructionDefn();

  dspram.accLoadShifter();

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = pixel0 , Acc = max

dspram.startInstructionDefn();

  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i + 0 , ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Acc = pixel0

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_B, comparisonPixelB , ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Mem = pixel0

compareMax( pixelNeighbourhoodA + 1 , 1 );
compareMax( pixelNeighbourhoodA + 2 , 0 );
compareMax( pixelNeighbourhoodA + 4 , 0 );
compareMax( pixelNeighbourhoodA + 4 , 1 );
compareMax( pixelNeighbourhoodA + 5 , 0 );
compareMax( pixelNeighbourhoodA + 7 , 0 );
compareMax( pixelNeighbourhoodA + 7 , 1 );
compareMax( pixelNeighbourhoodA + 8 , 0 );

// Store the maximum value
dspram.startInstructionDefn();

  dspram.disableByteSelect();
  dspram.multAxB( oneA , comparisonPixelB );

dspram.endInstructionDefn( DEBUG );
```

```
    // M0 = max

    dspram.startInstructionDefn();
    dspram.endInstructionDefn( DEBUG );
    // M0 = X, M1 = max

    dspram.startInstructionDefn();
    dspram.endInstructionDefn( DEBUG );
    // M0 = X, M1 = X, M2 = max

    dspram.startInstructionDefn();

      dspram.accLoadShifter();

    dspram.endInstructionDefn( DEBUG );
    // M0 = X, M1 = X, M2 = X, Acc = max

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i + 1, ACC_0 );

    dspram.endInstructionDefn( DEBUG );
    // M0 = X, M1 = X, M2 = X, Mem = max

#endif

#if DEBUG

    abort();

#endif

  } // for

} // dilate
```

## Listing D.8: Erode

```
// $Id: erode.cc,v 1.1 2002/07/29 20:31:57 dillen Exp dillen $
//
// $Log: erode.cc,v $
// Revision 1.1  2002/07/29 20:31:57  dillen
// Initial revision
//

#include <iostream>
#include <dspramClass.h>
#include "../image/image.h"

void image::erode( void )
{

  int i;
  int offset;

  // Step 1: Shift pixels to left and right neighbours
  // Step 2: Find the largest pixel value for lower pixel
  // Step 3: Store the lower pixel value in BANK B
  // Step 4: Find the smallest pixel value for the upper pixel
  // Step 5: Store the upper pixel in BANK B

  // Calculate the offset
  offset = width / (numberOfPEs * 2);

  for ( i = offset; i < (columnImageSize - offset); i++ )
  {

    collectNeighbours( i );

#if DEBUG

    dumpNeighbourhood( 1 );

#endif

#ifdef ENHANCED_ROUTING

    // Store the first neighbour into the maximum location
    dspram.startInstructionDefn();

      dspram.enableByteSelect( 1 );
      dspram.saveInputA ( BANK_A, pixelNeighbourhoodA );

    dspram.endInstructionDefn(DEBUG);

    dspram.startInstructionDefn();

      dspram.memLoadInputA ( BANK_B, comparisonPixelB );

    dspram.endInstructionDefn(DEBUG);
```

```
    compareMin( pixelNeighbourhoodA + 1, 0 );
    compareMin( pixelNeighbourhoodA + 1, 1 );
    compareMin( pixelNeighbourhoodA + 3, 1 );
    compareMin( pixelNeighbourhoodA + 4, 0 );
    compareMin( pixelNeighbourhoodA + 4, 1 );
    compareMin( pixelNeighbourhoodA + 6, 1 );
    compareMin( pixelNeighbourhoodA + 7, 0 );
    compareMin( pixelNeighbourhoodA + 7, 1 );

    // Save the maximum value to address B
    dspram.startInstructionDefn();

      dspram.disableByteSelect();
      dspram.saveInputA( BANK_B, comparisonPixelB );

    dspram.endInstructionDefn(DEBUG);

    dspram.startInstructionDefn();

      dspram.memLoadInputA( BANK_B, imageAddressB + 2 * i + 0 );

    dspram.endInstructionDefn(DEBUG);

    // Process the upper pixel
    dspram.startInstructionDefn();

      dspram.enableByteSelect( 0 );
      dspram.saveInputA( BANK_A, pixelNeighbourhoodA + 1 );

    dspram.endInstructionDefn(DEBUG);

    dspram.startInstructionDefn();

      dspram.memLoadInputA( BANK_B, comparisonPixelB );

    dspram.endInstructionDefn(DEBUG);

    compareMin( pixelNeighbourhoodA + 1, 1 );
    compareMin( pixelNeighbourhoodA + 2, 0 );
    compareMin( pixelNeighbourhoodA + 4, 0 );
    compareMin( pixelNeighbourhoodA + 4, 1 );
    compareMin( pixelNeighbourhoodA + 5, 0 );
    compareMin( pixelNeighbourhoodA + 7, 0 );
    compareMin( pixelNeighbourhoodA + 7, 1 );
    compareMin( pixelNeighbourhoodA + 8, 0 );

    // Save the maximum value to address B
    dspram.startInstructionDefn();

      dspram.disableByteSelect();
      dspram.saveInputA( BANK_B, comparisonPixelB );

    dspram.endInstructionDefn(DEBUG);

    dspram.startInstructionDefn();

      dspram.memLoadInputA( BANK_B, imageAddressB + 2 * i + 1 );

    dspram.endInstructionDefn(DEBUG);

#else

    // Store the first neighbour into the maximum location
    dspram.startInstructionDefn();

      dspram.disableAdder();
      dspram.setShifter( noShift );
      dspram.enableByteSelect( 1 );  // Upper byte
      dspram.multAxB( pixelNeighbourhoodA, oneB );

    dspram.endInstructionDefn( DEBUG );
    // M0 = p(0)

    // Nops
    dspram.startInstructionDefn();
    dspram.endInstructionDefn( DEBUG );
    // M0 = X, M1 = p(0)

    dspram.startInstructionDefn();
    dspram.endInstructionDefn( DEBUG );
    // M0 = X, M1 = X, M2 = p(0)

    dspram.startInstructionDefn();

      dspram.accLoadShifter();

    dspram.endInstructionDefn( DEBUG );
    // M0 = X, M1 = X, M2 = X, Acc = p(0)

    dspram.startInstructionDefn();
```

```
  dspram.memLoadAccWord( BANK_B, comparisonPixelB , ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Mem = p(0)

compareMin( pixelNeighbourhoodA + 1 , 0 );
compareMin( pixelNeighbourhoodA + 1 , 1 );
compareMin( pixelNeighbourhoodA + 3 , 1 );
compareMin( pixelNeighbourhoodA + 4 , 0 );
compareMin( pixelNeighbourhoodA + 4 , 1 );
compareMin( pixelNeighbourhoodA + 6 , 1 );
compareMin( pixelNeighbourhoodA + 7 , 0 );
compareMin( pixelNeighbourhoodA + 7 , 1 );

// Store the maximum value
dspram.startInstructionDefn();

  dspram.disableByteSelect();
  dspram.multAxB( oneA , comparisonPixelB );

dspram.endInstructionDefn( DEBUG );
// M0 = max

dspram.startInstructionDefn();

  dspram.enableByteSelect( 0 );
  dspram.multAxB( pixelNeighbourhoodA + 1 , oneB );

dspram.endInstructionDefn( DEBUG );
// M0 = pixel0 , M1 = max

dspram.startInstructionDefn();
dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = pixel0 , M2 = max

dspram.startInstructionDefn();

  dspram.accLoadShifter();

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = pixel0 , Acc = max

dspram.startInstructionDefn();

  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i + 0 , ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Acc = pixel0

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_B, comparisonPixelB , ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Mem = pixel0

compareMin( pixelNeighbourhoodA + 1 , 1 );
compareMin( pixelNeighbourhoodA + 2 , 0 );
compareMin( pixelNeighbourhoodA + 4 , 0 );
compareMin( pixelNeighbourhoodA + 4 , 1 );
compareMin( pixelNeighbourhoodA + 5 , 0 );
compareMin( pixelNeighbourhoodA + 7 , 0 );
compareMin( pixelNeighbourhoodA + 7 , 1 );
compareMin( pixelNeighbourhoodA + 8 , 0 );

// Store the maximum value
dspram.startInstructionDefn();

  dspram.disableByteSelect();
  dspram.multAxB( oneA , comparisonPixelB );

dspram.endInstructionDefn( DEBUG );
// M0 = max

dspram.startInstructionDefn();
dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = max

dspram.startInstructionDefn();
dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = max

dspram.startInstructionDefn();

  dspram.accLoadShifter();

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Acc = max

dspram.startInstructionDefn();
```

```
      dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i + 1, ACC_0 );

    dspram.endInstructionDefn( DEBUG );
    // M0 = X, M1 = X, M2 = X, Mem = max

#endif // ENHANCED_ROUTING

#if DEBUG

    abort();

#endif

  } // for

} // dilate
```

## Listing D.9: Median Filter

```
// $Id: medianFilter.cc,v 1.2 2002/08/01 16:06:30 dillen Exp dillen $
//
// $Log: medianFilter.cc,v $
// Revision 1.2  2002/08/01 16:06:30  dillen
// Finished median filter algorithm
//
// Revision 1.1  2002/07/29 21:39:07  dillen
// Initial revision
//

#include <iostream>
#include <dspramClass.h>
#include "../image/image.h"

void image::medianFilter( void )
{

  int i;
  int offset;

  // Step 1: Shift pixels to left and right neighbours
  // Step 2: Sort lower pixel neighbourhood
  // Step 3: Store median value
  // Step 4: Sort upper pixel neighbourhood
  // Step 5: Store median value

  // Calculate the offset
  offset = width / (numberOfPEs * 2);

  for ( i = offset; i < (columnImageSize - offset); i++ )
  {

    collectNeighbours( i );

#if DEBUG

    dumpNeighbourhood( 1 );

#endif

    // Copy lower neighbourhood pixels to B memory bank
    dspram.startInstructionDefn();

      dspram.injectOntoBroadcastBus(1);
      dspram.regLoadBroadcastBus();

    dspram.endInstructionDefn( DEBUG );
    // R = 1

    dspram.startInstructionDefn();

      dspram.enableByteSelect(1);
      dspram.multAxR( pixelNeighbourhoodA + 0 );
      dspram.disableAdder();
      dspram.setShifter( noShift );

    dspram.endInstructionDefn( DEBUG );
    // M0 = UL

    dspram.startInstructionDefn();

      dspram.enableByteSelect(0);
      dspram.multAxR( pixelNeighbourhoodA + 1 );

    dspram.endInstructionDefn( DEBUG );
    // M0 = UM, M1 = UL

    dspram.startInstructionDefn();

      dspram.enableByteSelect(1);
      dspram.multAxR( pixelNeighbourhoodA + 1 );
```

```
dspram.endInstructionDefn( DEBUG );
// M0 = UR, M1 = UM, M2 = UL

dspram.startInstructionDefn();

  dspram.enableByteSelect(1);
  dspram.multAxR( pixelNeighbourhoodA + 3 );
  dspram.accLoadShifter();

dspram.endInstructionDefn( DEBUG );
// M0 = ML, M1 = UR, M2 = UM, Acc = UL

dspram.startInstructionDefn();

  dspram.enableByteSelect(0);
  dspram.multAxR( pixelNeighbourhoodA + 4 );
  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, sortingArrayB + 0, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = MM, M1 = ML, M2 = UR, Acc = UM, Mem = UL

dspram.startInstructionDefn();

  dspram.enableByteSelect(1);
  dspram.multAxR( pixelNeighbourhoodA + 4 );
  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, sortingArrayB + 1, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = MR, M1 = MM, M2 = ML, Acc = UR, Mem = UM

dspram.startInstructionDefn();

  dspram.enableByteSelect(1);
  dspram.multAxR( pixelNeighbourhoodA + 5 );
  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, sortingArrayB + 2, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = LL, M1 = MR, M2 = MM, Acc = ML, Mem = UR

dspram.startInstructionDefn();

  dspram.enableByteSelect(0);
  dspram.multAxR( pixelNeighbourhoodA + 7 );
  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, sortingArrayB + 3, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = LM, M1 = LL, M2 = MR, Acc = MM, Mem = ML

dspram.startInstructionDefn();

  dspram.enableByteSelect(1);
  dspram.multAxR( pixelNeighbourhoodA + 7 );
  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, sortingArrayB + 4, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = LR, M1 = LM, M2 = LL, Acc = MR, Mem = MM

dspram.startInstructionDefn();

  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, sortingArrayB + 5, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = LR, M2 = LM, Acc = LL, Mem = MR

dspram.startInstructionDefn();

  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, sortingArrayB + 6, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = LR, Acc = LM, Mem = LL

dspram.startInstructionDefn();

  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, sortingArrayB + 7, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = X, M1 = X, M2 = X, Acc = LR, Mem = LM

dspram.startInstructionDefn();

  dspram.memLoadAccWord( BANK_B, sortingArrayB + 8, ACC_0 );

dspram.endInstructionDefn( DEBUG );
```

```
// M0 = X, M1 = X, M2 = X, Acc = X, Mem = LR

sortNeighbourhood();

// Store the median value into the image B bank
dspram.startInstructionDefn();

  dspram.injectOntoBroadcastBus(1);
  dspram.regLoadBroadcastBus();
  dspram.disableByteSelect();
  dspram.multAxB( oneA, sortingArrayB + 4 );

dspram.endInstructionDefn();
// M0 = pixel

dspram.startInstructionDefn();

  dspram.enableByteSelect(0);
  dspram.multAxR( pixelNeighbourhoodA + 1 );
  dspram.disableAdder();
  dspram.setShifter( noShift );

dspram.endInstructionDefn();
// M0 = UL, M1 = pixel

dspram.startInstructionDefn();

  dspram.enableByteSelect(1);
  dspram.multAxR( pixelNeighbourhoodA + 1 );

dspram.endInstructionDefn();
// M0 = UM, M1 = UL, M2 = pixel

dspram.startInstructionDefn();

  dspram.enableByteSelect(0);
  dspram.multAxR( pixelNeighbourhoodA + 2 );
  dspram.accLoadShifter();

dspram.endInstructionDefn();
// M0 = UR, M1 = UM, M2 = UL, Acc = pixel

dspram.startInstructionDefn();

  dspram.enableByteSelect(0);
  dspram.multAxR( pixelNeighbourhoodA + 4 );
  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i + 0, ACC_0 );

dspram.endInstructionDefn();
// M0 = ML, M1 = UR, M2 = UM, Acc = UL, Mem = pixel

// Finish Copy upper neighbourhood pixels to B memory bank
dspram.startInstructionDefn();

  dspram.enableByteSelect(1);
  dspram.multAxR( pixelNeighbourhoodA + 4 );
  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, sortingArrayB + 0, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = MM, M1 = ML, M2 = UR, Acc = UM, Mem = UL

dspram.startInstructionDefn();

  dspram.enableByteSelect(0);
  dspram.multAxR( pixelNeighbourhoodA + 5 );
  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, sortingArrayB + 1, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = MR, M1 = MM, M2 = ML, Acc = UR, Mem = UM

dspram.startInstructionDefn();

  dspram.enableByteSelect(0);
  dspram.multAxR( pixelNeighbourhoodA + 7 );
  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, sortingArrayB + 2, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = LL, M1 = MR, M2 = MM, Acc = ML, Mem = UR

dspram.startInstructionDefn();

  dspram.enableByteSelect(1);
  dspram.multAxR( pixelNeighbourhoodA + 7 );
  dspram.accLoadShifter();
  dspram.memLoadAccWord( BANK_B, sortingArrayB + 3, ACC_0 );

dspram.endInstructionDefn( DEBUG );
// M0 = LM, M1 = LL, M2 = MR, Acc = MM, Mem = ML
```

```
    dspram.startInstructionDefn();

      dspram.enableByteSelect(0);
      dspram.multAxR( pixelNeighbourhoodA + 8 );
      dspram.accLoadShifter();
      dspram.memLoadAccWord( BANK_B, sortingArrayB + 4, ACC_0 );

    dspram.endInstructionDefn( DEBUG );
    // M0 = LR, M1 = LM, M2 = LL, Acc = MR, Mem = MM

    dspram.startInstructionDefn();

      dspram.accLoadShifter();
      dspram.memLoadAccWord( BANK_B, sortingArrayB + 5, ACC_0 );

    dspram.endInstructionDefn( DEBUG );
    // M0 = X, M1 = LR, M2 = LM, Acc = LL, Mem = MR

    dspram.startInstructionDefn();

      dspram.accLoadShifter();
      dspram.memLoadAccWord( BANK_B, sortingArrayB + 6, ACC_0 );

    dspram.endInstructionDefn( DEBUG );
    // M0 = X, M1 = X, M2 = LR, Acc = LM, Mem = LL

    dspram.startInstructionDefn();

      dspram.accLoadShifter();
      dspram.memLoadAccWord( BANK_B, sortingArrayB + 7, ACC_0 );

    dspram.endInstructionDefn( DEBUG );
    // M0 = X, M1 = X, M2 = X, Acc = LR, Mem = LM

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_B, sortingArrayB + 8, ACC_0 );

    dspram.endInstructionDefn( DEBUG );
    // M0 = X, M1 = X, M2 = X, Acc = X, Mem = LR

    // Sort the pixel neighbourhood
    sortNeighbourhood();

    // Store the median value into the image B bank
    dspram.startInstructionDefn();

      dspram.disableByteSelect();
      dspram.multAxB( oneA, sortingArrayB + 4 );

    dspram.endInstructionDefn();
    // M0 = pixel

    dspram.startInstructionDefn();
    dspram.endInstructionDefn();
    // M0 = X, M1 = pixel

    dspram.startInstructionDefn();
    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = pixel

    dspram.startInstructionDefn();

      dspram.accLoadShifter();

    dspram.endInstructionDefn();
    // M0 = X, M1 = X, M2 = X, Acc = pixel

    dspram.startInstructionDefn();

      dspram.memLoadAccWord( BANK_B, imageAddressB + 2 * i + 1, ACC_0 );

    dspram.endInstructionDefn();
    // Mem = pixel

#if DEBUG

    abort();

#endif

  } // for

} // medianFilter
```